Introduction to Running Time Analysis of Algorithms

Dimitris Diochnos

January 26, 2018

CS 251 Data Structures
UIC

Outline

1. Definitions
2. Examples

Definitions

Basics

We want to analyze the running time behavior $T(N)$ of algorithms.

- N is a positive integer capturing the input size of a particular instance where the algorithm is applied.
  (e.g., N is the size of an array to be sorted.)

Basics

We want to analyze the running time behavior $T(N)$ of algorithms.

- N is a positive integer capturing the input size of a particular instance where the algorithm is applied.
  (e.g., N is the size of an array to be sorted.)

In other words, $T(N)$ is a function,

$$T(N) : \mathbb{N}^* \mapsto \mathbb{N}^*$$

- The domain reflects the different values that the problem size can take.
- The range is the running time (in discrete units of time).

Upper Bounds

Definition (Big O - Upper Bound)

$T(N) = O(f(N))$ if there exist positive constants c and n_0 such that

$$T(N) \leq cf(N)$$

for every $N \geq n_0$.

Upper Bounds

Definition (Big O - Upper Bound)

$T(N) = O(f(N))$ if there exist positive constants c and n_0 such that

$$T(N) \leq cf(N)$$

for every $N \geq n_0$.

Example

$3N^2 = O(N^3)$

($c = 1$ and $n_0 = 3$)
($c = 2$ and $n_0 = 2$)
($c = 3$ and $n_0 = 1$)
(one is enough; constants do not matter!)

Upper Bounds

Definition (Big O - Upper Bound)

$T(N) = O(f(N))$ if there exist positive constants c and n_0 such that

$$T(N) \leq cf(N)$$

for every $N \geq n_0$.

Example

$3N^2 = O(N^3)$

$(c = 1 \text{ and } n_0 = 3)$
$(c = 2 \text{ and } n_0 = 2)$
$(c = 3 \text{ and } n_0 = 1)$
(one is enough; constants do not matter!)

We say,

- $3N^2$ is big-oh N to the third, or that,
- $3N^2$ is order of N to the third.

Upper Bounds

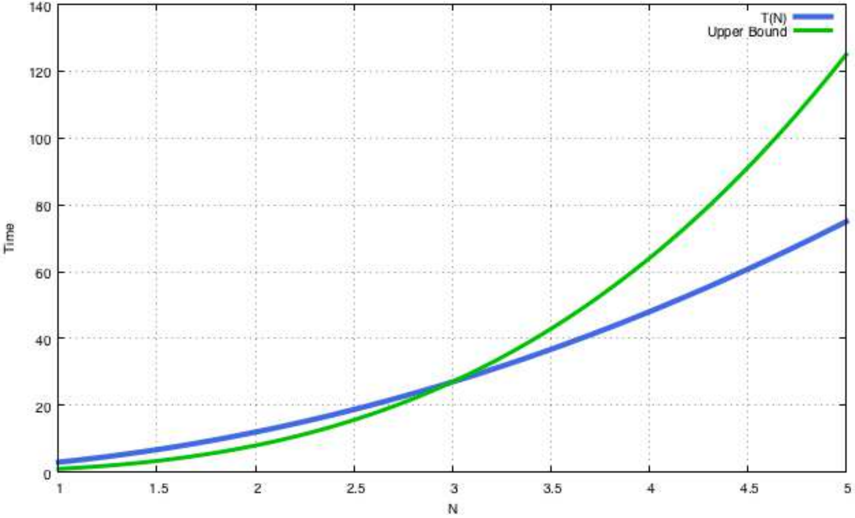
Definition (Set Definition of Big O)

$$O(f(N)) = \{g(N) \quad : \quad \text{there exist } c, n_0 > 0 \text{ such that} \\ 0 \leq g(N) \leq cf(N) \text{ for all } N \geq n_0\}$$

Example

$3N^2 \in O(N^3)$

An Example for the O Notation



$$T(N) = 3N^2 \leq N^3$$

Lower Bounds

Definition (Big Ω - Lower Bound)

$T(N) = \Omega(f(N))$ if there are positive constants c and n_0 such that

$$cf(N) \leq T(N)$$

for every $N \geq n_0$.

Example

$\sqrt{N} = \Omega\left(\log_2 N\right)$ $\qquad$ ($c = 1$ and $n_0 = 16$)

Lower Bounds

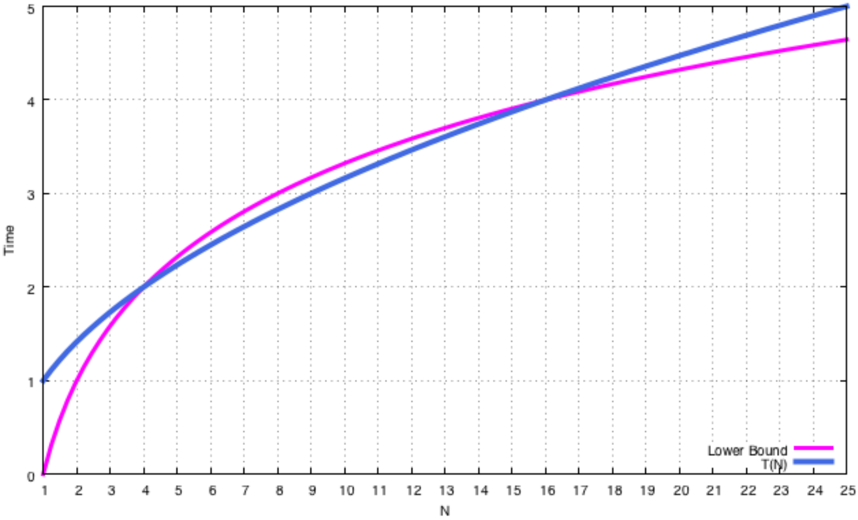
Definition (Set Definition of Big Ω)

$$\Omega(f(n)) = \{g(n) \quad : \quad \text{there exist } c, n_0 > 0 \text{ such that } 0 \leq cf(n) \leq g(n) \text{ for all } n \geq n_0\}$$

Example

$\sqrt{N} \in \Omega(\log_2 N)$

The Example for the Ω Notation



$$\log_2 N \leq T(N) = \sqrt{N}$$

Tight Bounds

Definition

$T(N) = \Theta(f(N))$ if and only if

$$\begin{cases} T(N) = \Omega(f(N)) & \text{and} \\ T(N) = O(f(N)) \end{cases}$$

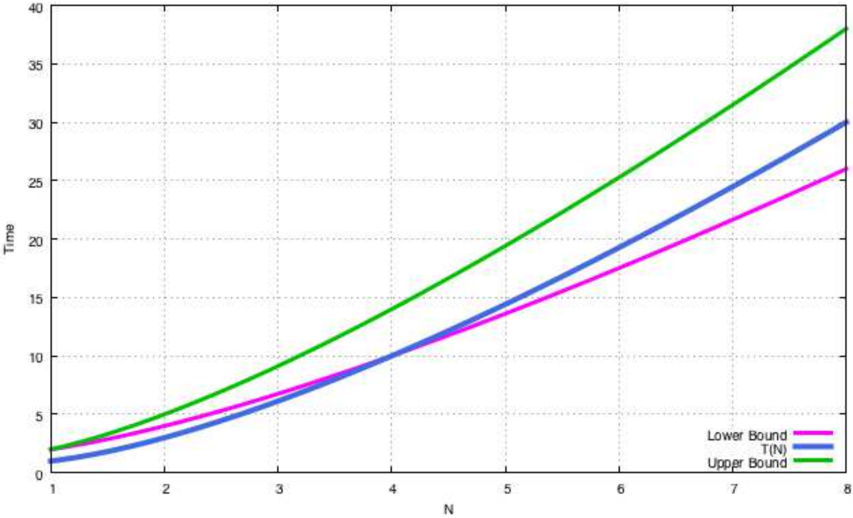
Example

$\frac{3}{2}N\log_2 N - N + 2 = \Theta\left(N\log_2 N\right)$

$$N\log_2 N + 2 \leq \frac{3}{2}N\log_2 N - N + 2 \leq \frac{3}{2}N\log_2 N + 2$$

for every $N \geq 4$.

The Example for the Θ Notation



$$N\log_2 N + 2 \leq T(N) = \frac{3}{2}N\log_2(N) - N + 2 \leq \frac{3}{2}N\log_2(N)$$

Some General Rules

Rule 1 Let $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$. Then,

- $T_1(N) + T_2(N) = O(f(N) + g(N))$
  (intuitively, $T_1 + T_2 = O(\max(f(N), g(N)))$)
- $T_1(N) \cdot T_2(N) = O(f(N) \cdot g(N))$

Some General Rules

Rule 1 Let $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$. Then,

- $T_1(N) + T_2(N) = O(f(N) + g(N))$
  (intuitively, $T_1 + T_2 = O(\max(f(N), g(N)))$)
- $T_1(N) \cdot T_2(N) = O(f(N) \cdot g(N))$

Rule 2 Let $T(N)$ be a polynomial of degree k. Then, $T(N) = \Theta(n^k)$.

Some General Rules

Rule 1 Let $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$. Then,

- $T_1(N) + T_2(N) = O(f(N)+g(N))$
  (intuitively, $T_1 + T_2 = O(\max(f(N), g(N)))$)
- $T_1(N) \cdot T_2(N) = O(f(N) \cdot g(N))$

Rule 2 Let $T(N)$ be a polynomial of degree k. Then, $T(N) = \Theta(n^k)$.

Rule 3 $\log^k(N) = O(N)$ for any constant k.

Typical Growth Rates

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Examples

Constant Time Operations

Definition (Execution in Constant Time)

An operation (more generally an algorithm) is said to be constant time, written as $O(1)$ time, if the **time** required **to execute** the operation (or the algorithm) is upper bounded by a value that **does not depend on the size of the input**.

Constant Time Operations

Definition (Execution in Constant Time)

An operation (more generally an algorithm) is said to be constant time, written as $O(1)$ time, if the **time** required **to execute** the operation (or the algorithm) is upper bounded by a value that **does not depend on the size of the input**.

Examples

- Basic arithmetic operations (addition, subtraction, multiplication, division, exponentiation) between two numbers.***

Constant Time Operations

Definition (Execution in Constant Time)

An operation (more generally an algorithm) is said to be constant time, written as $O(1)$ time, if the **time** required **to execute** the operation (or the algorithm) is upper bounded by a value that **does not depend on the size of the input**.

Examples

- Basic arithmetic operations (addition, subtraction, multiplication, division, exponentiation) between two numbers.***
- Any comparison $(<, =, >)$ between two numbers.***

Constant Time Operations

Definition (Execution in Constant Time)

An operation (more generally an algorithm) is said to be constant time, written as $O(1)$ time, if the **time** required **to execute** the operation (or the algorithm) is upper bounded by a value that **does not depend on the size of the input**.

Examples

- Basic arithmetic operations (addition, subtraction, multiplication, division, exponentiation) between two numbers.***
- Any comparison ($<, =, >$) between two numbers.***
- Accessing a particular element in an array; e.g., $a[5]$.

Constant Time Operations

Definition (Execution in Constant Time)

An operation (more generally an algorithm) is said to be constant time, written as $O(1)$ time, if the **time** required **to execute** the operation (or the algorithm) is upper bounded by a value that **does not depend on the size of the input**.

Examples

- Basic arithmetic operations (addition, subtraction, multiplication, division, exponentiation) between two numbers.***
- Any comparison $(<, =, >)$ between two numbers.***
- Accessing a particular element in an array; e.g., $a[5]$.
- Swap two elements in an array if necessary.

Constant Time Operations

Definition (Execution in Constant Time)

An operation (more generally an algorithm) is said to be constant time, written as $O(1)$ time, if the **time** required **to execute** the operation (or the algorithm) is upper bounded by a value that **does not depend on the size of the input**.

Examples

- Basic arithmetic operations (addition, subtraction, multiplication, division, exponentiation) between two numbers.***
- Any comparison ($<, =, >$) between two numbers.***
- Accessing a particular element in an array; e.g., $a[5]$.
- Swap two elements in an array if necessary.
- Assign a value to a pointer.

Constant Time Operations

Definition (Execution in Constant Time)

An operation (more generally an algorithm) is said to be constant time, written as $O(1)$ time, if the **time** required **to execute** the operation (or the algorithm) is upper bounded by a value that **does not depend on the size of the input**.

Examples

- Basic arithmetic operations (addition, subtraction, multiplication, division, exponentiation) between two numbers.***
- Any comparison ($<, =, >$) between two numbers.***
- Accessing a particular element in an array; e.g., $a[5]$.
- Swap two elements in an array if necessary.
- Assign a value to a pointer.
- Find the minimum value in a sorted array; $a[0]$.

Constant Time Operations

Definition (Execution in Constant Time)

An operation (more generally an algorithm) is said to be constant time, written as $O(1)$ time, if the **time** required **to execute** the operation (or the algorithm) is upper bounded by a value that **does not depend on the size of the input**.

Examples

- Basic arithmetic operations (addition, subtraction, multiplication, division, exponentiation) between two numbers.***
- Any comparison $(<, =, >)$ between two numbers.***
- Accessing a particular element in an array; e.g., $a[5]$.
- Swap two elements in an array if necessary.
- Assign a value to a pointer.
- Find the minimum value in a sorted array; $a[0]$.

*** numbers stored in registers.

Insert a Node in a Doubly Linked List

```
LIST-INSERT (List L, Listnode x)
x.next = L.head
if (L.head != NULL)
    (L.head).prev = x
L.head = x
x.prev = NULL
```

Insert a Node in a Doubly Linked List

```
LIST-INSERT (List L, Listnode x)
1  x.next = L.head
2  if (L.head != NULL)
3      (L.head).prev = x
4  L.head = x
5  x.prev = NULL
```

- Line 3: 1 time step (1 assignment)
- Line 2: 1 time step (1 comparison)
- Lines 2-3: At most 2 time steps
- Lines 1, 4, and 5: 1 time step each

Total time is at most 5 time steps; so $O(1)$.

Find a Node with a Particular Value in a List

```
LIST-FIND (List L, Listvalue v)
 current = L.head
while ((current != NULL) and (current.val != v))
    current = current.next
 return current
```

Find a Node with a Particular Value in a List

```
LIST-FIND (List L, Listvalue v)
1  current = L.head
2  while ((current != NULL) and (current.val != v))
3      current = current.next
4  return current
```

Worst case scenario (v not in the list)

- Line 3: 1 time step $\Rightarrow$ Total: N
- Line 2: Total $2N + 1$ time steps
- Lines 1 and 4: 1 time step each $\Rightarrow$ Total: 2

Total time is at most $3N + 3$ time steps; so $O(N)$.

Compute the Sum $\sum_{i=1}^{N} i^3$

```
int sum( int n )
{
    int partialSum;

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

Compute the Sum $\sum_{i=1}^{N} i^3$

```
  int sum( int n )
  {
      int partialSum;

1     partialSum = 0;
2     for( int i = 1; i <= n; ++i )
3         partialSum += i * i * i;
4     return partialSum;
  }
```

- Lines 1 and 4: 1 unit of time each $\Rightarrow$ Total: 2
- Line 3: 4 units of time (2 multiplications, 1 addition, 1 assignment) $\Rightarrow$ Total: $4N$
- Line 2: $\underbrace{1}_{init} + \underbrace{(N+1)}_{comparisons} + \underbrace{N}_{increments} \Rightarrow$ Total: $2N + 2$

Therefore, the overall total is $6N + 4$ time steps. (O(N))

General Rules for Obtaining Upper Bounds

Rule 1 – FOR Loops The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.

General Rules for Obtaining Upper Bounds

Rule 1 – FOR Loops The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.

Rule 2 – Nested Loops Analyze inside out, applying Rule 1 in every case.

General Rules for Obtaining Upper Bounds

Rule 1 – FOR Loops The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.

Rule 2 – Nested Loops Analyze inside out, applying Rule 1 in every case.

Rule 3 – Consecutive Statements Add up the various components.

General Rules for Obtaining Upper Bounds

Rule 1 – FOR Loops The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.

Rule 2 – Nested Loops Analyze inside out, applying Rule 1 in every case.

Rule 3 – Consecutive Statements Add up the various components.

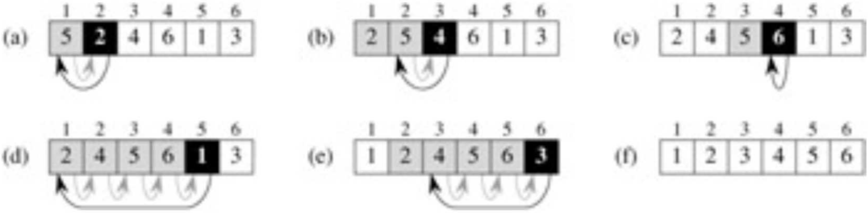
Rule 4 – If/Else In a situation as below:

```
if ( condition )
    S1
else
    S2
```

the running time is upper bounded by the running time of the test plus the larger of the running times of S1 and S2.

Insertion Sort

```
INSERTION-SORT(A)
  for j ← 2 to length[A]
       do key ← A[j]
          ▹ Insert A[j] into the sorted sequence A[1 □ j - 1].
          i ← j - 1
          while i > 0 and A[i] > key
              do A[i + 1] ← A[i]
                 i ← i - 1
          A[i + 1] ← key
```



Insertion Sort - Best Case (Input Array Sorted)

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▹ Insert A[j] into the sorted sequence A[1 □ j - 1].
4          i ← j - 1
5          while i > 0 and A[i] > key
6              do A[i + 1] ← A[i]
7                 i ← i - 1
8          A[i + 1] ← key
```

- Line 5: 2 comparisons that always fail ⇒ Ignore lines 6 and 7. ⇒ $O(1)$ time steps per iteration

Insertion Sort - Best Case (Input Array Sorted)

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▹ Insert A[j] into the sorted sequence A[1 □ j - 1].
4          i ← j - 1
5          while i > 0 and A[i] > key
6              do A[i + 1] ← A[i]
7                 i ← i - 1
8          A[i + 1] ← key
```

- Line 5: 2 comparisons that always fail $\Rightarrow$ Ignore lines 6 and 7. $\Rightarrow$ $O(1)$ time steps per iteration
- Lines 2, 4, 8: $O(1)$ time steps per iteration

Insertion Sort - Best Case (Input Array Sorted)

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▹ Insert A[j] into the sorted sequence A[1 □ j - 1].
4          i ← j - 1
5          while i > 0 and A[i] > key
6              do A[i + 1] ← A[i]
7                 i ← i - 1
8          A[i + 1] ← key
```

- Line 5: 2 comparisons that always fail ⇒ Ignore lines 6 and 7. ⇒ $O(1)$ time steps per iteration
- Lines 2, 4, 8: $O(1)$ time steps per iteration
- Line 1: At most 3 operations per iteration ⇒ $O(1)$ time per iteration (Overall: 1 init, N comparisons, $(N - 1)$ increments)

Insertion Sort - Best Case (Input Array Sorted)

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▹ Insert A[j] into the sorted sequence A[1 □ j - 1].
4          i ← j - 1
5          while i > 0 and A[i] > key
6              do A[i + 1] ← A[i]
7                 i ← i - 1
8          A[i + 1] ← key
```

- Line 5: 2 comparisons that always fail ⇒ Ignore lines 6 and 7. ⇒ $O(1)$ time steps per iteration
- Lines 2, 4, 8: $O(1)$ time steps per iteration
- Line 1: At most 3 operations per iteration ⇒ $O(1)$ time per iteration (Overall: 1 init, N comparisons, $(N-1)$ increments)

By the **rule of FOR loops**, since the body of the FOR loop will be executed $N-1$ times, we have that the total running time is $O(N)$.
(We also applied the **rule of nested loops** - but it was trivial here.)

Insertion Sort - Worst Case (Input Array Inv. Sorted)

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▹ Insert A[j] into the sorted sequence A[1 □ j - 1].
4          i ← j - 1
5          while i > 0 and A[i] > key
6              do A[i + 1] ← A[i]
7                 i ← i - 1
8          A[i + 1] ← key
```

- **While** loop (lines 5 - 7): Every iteration in $O(1)$ time. Loop at most $j-1$ times $\Rightarrow$ Total time at most $c(j-1)+c = cj$ time steps (per j)

Insertion Sort - Worst Case (Input Array Inv. Sorted)

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▹ Insert A[j] into the sorted sequence A[1 □ j - 1].
4          i ← j - 1
5          while i > 0 and A[i] > key
6              do A[i + 1] ← A[i]
7                 i ← i - 1
8          A[i + 1] ← key
```

- **While** loop (lines 5 - 7): Every iteration in $O(1)$ time. Loop at most $j - 1$ times $\Rightarrow$ Total time at most $c(j - 1) + c = cj$ time steps (per j)
- Lines 2, 4, 8: $O(1)$ time.

Insertion Sort - Worst Case (Input Array Inv. Sorted)

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▹ Insert A[j] into the sorted sequence A[1 □ j - 1].
4          i ← j - 1
5          while i > 0 and A[i] > key
6              do A[i + 1] ← A[i]
7                 i ← i - 1
8          A[i + 1] ← key
```

- **While** loop (lines 5 - 7): Every iteration in $O(1)$ time. Loop at most $j-1$ times $\Rightarrow$ Total time at most $c(j-1)+c = cj$ time steps (per j)
- Lines 2, 4, 8: $O(1)$ time.
- Line 1: $O(1)$ time per iteration

Insertion Sort - Worst Case (Input Array Inv. Sorted)

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▹ Insert A[j] into the sorted sequence A[1 □ j - 1].
4          i ← j - 1
5          while i > 0 and A[i] > key
6              do A[i + 1] ← A[i]
7                 i ← i - 1
8          A[i + 1] ← key
```

- **While** loop (lines 5 - 7): Every iteration in $O(1)$ time. Loop at most $j-1$ times $\Rightarrow$ Total time at most $c(j-1)+c = cj$ time steps (per j)
- Lines 2, 4, 8: $O(1)$ time.
- Line 1: $O(1)$ time per iteration

$$\begin{aligned}\text{Total time } T(N) &\leq c' + \sum_{j=2}^{N}(c' + cj) = c' + c'(N-1) + c\sum_{j=2}^{N} j \\ &= c'N + c\left(\frac{N(N+1)}{2} - 1\right) = O\left(N^2\right).\end{aligned}$$

Binary Search

What is the idea of binary search?

Binary Search

```
/**
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found or -1 if not found.
 */
template <typename Comparable>
int binarySearch( const vector<Comparable> & a, const Comparable & x )
{
    int low = 0, high = a.size( ) - 1;

    while( low <= high )
    {
        int mid = ( low + high ) / 2;

        if( a[ mid ] < x )
            low = mid + 1;
        else if( a[ mid ] > x )
            high = mid - 1;
        else
            return mid;   // Found
    }
    return NOT_FOUND;     // NOT_FOUND is defined as -1
}
```

Binary Search - Worst Case

Assumption: Let the array size be $N = 2^k$ for some $k \in \mathbb{N}^*$.

- Worst case: the number we are looking for is not in the array!
- In each iteration, we are left with an array that has size at most half of the previous array.
- For example,

1	5	7	10	15	20	27	33

Binary Search - Worst Case

Assumption: Let the array size be $N = 2^k$ for some $k \in \mathbb{N}^*$.

- Worst case: the number we are looking for is not in the array!
- In each iteration, we are left with an array that has size at most half of the previous array.
- For example,

$\cancel{1}$	$\cancel{5}$	$\cancel{7}$	$\cancel{10}$	15	20	27	33

Binary Search - Worst Case

Assumption: Let the array size be $N = 2^k$ for some $k \in \mathbb{N}^*$.

- Worst case: the number we are looking for is not in the array!
- In each iteration, we are left with an array that has size at most half of the previous array.
- For example, | $\cancel{1}$ | $\cancel{5}$ | $\cancel{7}$ | $\cancel{10}$ | 15 | 20 | 27 | 33 |
- So, we have

Queries	Array Size
0	N
1	$N/2$
2	$(N/2)/2$
$\vdots$	$\vdots$
q	$N/2^q$

Binary Search - Worst Case

Assumption: Let the array size be $N = 2^k$ for some $k \in \mathbb{N}^*$.

- Worst case: the number we are looking for is not in the array!
- In each iteration, we are left with an array that has size at most half of the previous array.
- For example,

~~1~~	~~5~~	~~7~~	~~10~~	15	20	27	33

- So, we have

Queries	Array Size
0	N
1	$N/2$
2	$(N/2)/2$
$\vdots$	$\vdots$
q	$N/2^q$

- Stop when $N/2^q < 1 \Rightarrow 2^q > N \Rightarrow q > \log_2(N) \Rightarrow q = 1 + k$.

Total time $O(\log_2(N))$

An Observation for Binary Search

Binary search splits the array ($N \geq 2$) into

$$\left(\lfloor N/2 \rfloor - 1\right) + 1 + \lceil N/2 \rceil$$

An Observation for Binary Search

Binary search splits the array ($N \geq 2$) into

$$\left(\lfloor N/2 \rfloor - 1\right) + 1 + \lceil N/2 \rceil$$

$\Rightarrow$ Worst case: N is odd, we are left with $\lceil N/2 \rceil$ (slightly more than $N/2$).

An Observation for Binary Search

Binary search splits the array ($N \geq 2$) into

$$(\lfloor N/2 \rfloor - 1) + 1 + \lceil N/2 \rceil$$

$\Rightarrow$ Worst case: N is odd, we are left with $\lceil N/2 \rceil$ (slightly more than $N/2$).

Lemma

Let $2^{k-1} < N < 2^k$. Then, $2^{k-2} < \lceil N/2 \rceil \leq 2^{k-1}$.

An Observation for Binary Search

Binary search splits the array ($N \geq 2$) into

$$(\lfloor N/2 \rfloor - 1) + 1 + \lceil N/2 \rceil$$

$\Rightarrow$ Worst case: N is odd, we are left with $\lceil N/2 \rceil$ (slightly more than $N/2$).

Lemma

Let $2^{k-1} < N < 2^k$. Then, $2^{k-2} < \lceil N/2 \rceil \leq 2^{k-1}$.

Queries	Array Size	Array Size Upper Bound
0	N	$< 2^k$
1	$\lceil N/2 \rceil$	$\leq 2^{k-1}$
2	$\lceil \lceil N/2 \rceil /2 \rceil$	$\leq 2^{k-2}$
$\vdots$	$\vdots$	$\vdots$
q	$\lceil \cdots \lceil \lceil N/2 \rceil /2 \rceil \cdots \rceil /2 \rceil$	$\leq 2^{k-q}$

An Observation for Binary Search

Binary search splits the array ($N \geq 2$) into

$$\left(\lfloor N/2 \rfloor - 1\right) + 1 + \lceil N/2 \rceil$$

$\Rightarrow$ Worst case: N is odd, we are left with $\lceil N/2 \rceil$ (slightly more than $N/2$).

Lemma

Let $2^{k-1} < N < 2^k$. Then, $2^{k-2} < \lceil N/2 \rceil \leq 2^{k-1}$.

Queries	Array Size	Array Size Upper Bound
0	N	$< 2^k$
1	$\lceil N/2 \rceil$	$\leq 2^{k-1}$
2	$\lceil \lceil N/2 \rceil /2 \rceil$	$\leq 2^{k-2}$
$\vdots$	$\vdots$	$\vdots$
q	$\lceil \cdots \lceil \lceil N/2 \rceil /2 \rceil \cdots \rceil /2 \rceil$	$\leq 2^{k-q}$

- Stop when $2^{k-q} < 1 \Rightarrow q = 1 + k = 1 + \lceil \log_2(N) \rceil$.

Total time $O\left(\log_2(N)\right)$

Appendix: A Few More Comments

Little Oh

Definition (Little Oh)

$T(N) = o(f(N))$ if, for all positive constants c, there exists an $n_0 > 0$ such that

$$T(N) < cf(N)$$

when $N \geq n_0$.

(Informal: $T(N) = o(f(N))$ if $T(N) = O(f(N))$ and $T(N) \neq \Theta(f(N))$.)

Example

$2N^2 = o(N^3)$ $\left(n_0 = \frac{2}{c}\right)$

Little Oh

Definition (Little Oh)

$T(N) = o(f(N))$ if, for all positive constants c, there exists an $n_0 > 0$ such that

$$T(N) < cf(N)$$

when $N \geq n_0$.

(Informal: $T(N) = o(f(N))$ if $T(N) = O(f(N))$ and $T(N) \neq \Theta(f(N))$.)

Example

$2N^2 = o(N^3)$ $\left(n_0 = \frac{2}{c}\right)$

Definition (Set Definition of Little Oh)

$$o(f(n)) = \{g(n) \quad : \quad \text{for any constant } c > 0, \text{ there is a constant } n_0 > 0 \text{ such that } 0 \leq g(n) < cf(n) \text{ for all } n \geq n_0\}$$

Little Omega

Definition (Little Omega)

$T(N) = \omega(f(N))$ if, for all positive constants c, there exists an $n_0 > 0$ such that

$$T(N) > cf(N)$$

when $N \geq n_0$.

(Informal: $T(N) = \omega(f(N))$ if $T(N) = \Omega(f(N))$ and $T(N) \neq \Theta(f(N))$.)

Example

$2N^2 = o(N^3)$ $\left(n_0 = \frac{2}{c}\right)$

Little Omega

Definition (Little Omega)

$T(N) = \omega(f(N))$ if, for all positive constants c, there exists an $n_0 > 0$ such that

$$T(N) > cf(N)$$

when $N \geq n_0$.

(Informal: $T(N) = \omega(f(N))$ if $T(N) = \Omega(f(N))$ and $T(N) \neq \Theta(f(N))$.)

Example

$2N^2 = o(N^3)$ $\left(n_0 = \frac{2}{c}\right)$

Definition (Set Definition of Little Omega)

$$\omega(f(n)) = \{g(n) \quad : \quad \begin{array}{l} \text{for any constant } c > 0, \\ \text{there is a constant } n_0 > 0 \text{ such that} \\ 0 \leq cf(n) < g(n) \text{ for all } n \geq n_0\} \end{array}$$

Using Limits to Compute Relationships

For two functions f and g we can compute the limit,

$$L = \lim_{N \to \infty} \frac{f(N)}{g(N)}$$

using L'Hôpital's rule.

- $L = 0 \Rightarrow f(N) = o(g(N))$.
- $L = c \neq 0 \Rightarrow f(N) = \Theta(g(N))$.
- $L = \infty \Rightarrow g(N) = o(f(N))$.
- L does not exist $\Rightarrow$ There is no relation (will never happen in our context)

Macro Substitution

Convention: A set in a formula represents a function from the set.

Example

$$N^2 + O(N) = O(N^2)$$

means that for any $f(N) \in O(N)$:

$$N^2 + f(N) = g(N)$$

for some $g(N) \in O(N^2)$.