11

LECTURE 11

Analysis of Algorithms

ALGORITHM EFFICIENCY

- Intuitively we see that binary search is much faster than linear search, but how do we analyze the efficiency of algorithms formally?

BIG – O NOTATION

- The time required to solve a problem depends on more than only the number of operations it uses.
- The time also depends on the hardware and software used to run the program that implements the algorithm.
- On a supercomputer we might be able to solve a problem of size n a million times faster than we can on a PC.
- One of the advantages of using big-O notation, we do not have to worry about the hardware and software used to implement an algorithm.

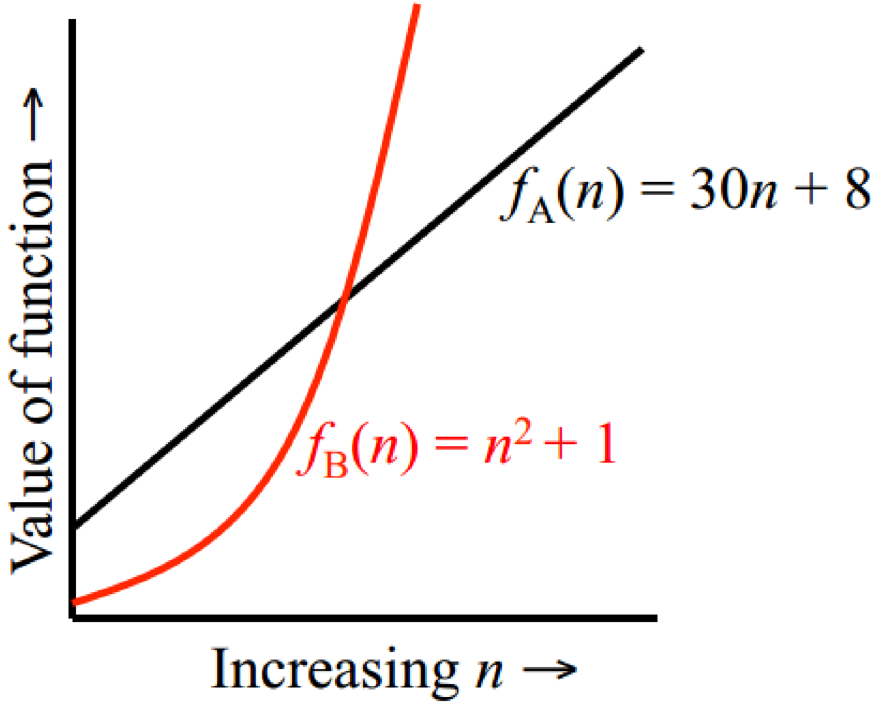
GROWTH OF FUNCTIONS

- Goal: To introduce the big-O notation and to show how to estimate the growth of functions using this notation and thereby to estimate the complexity (and hence the running time) of algorithms.

ORDERS OF GROWTH: MOTIVATION / VISUALIZATION

On a graph, as you go to the right, the faster growing function always eventually becomes the larger one...

- Suppose you are designing a web site to process user data (*e.g.*, financial records).
- Suppose database program A takes $f_A(n) = 30n + 8$ microseconds to process any n records, while program B takes $f_B(n) = n^2 + 1$ microseconds to process the n records.
- Which program do you choose, knowing you'll want to support millions of users?



CONCEPT OF ORDER OF GROWTH

- We say $f_A(n) = 30n + 8$ is (*at most*) *order of n*, or $O(n)$.
  - It is, at most, roughly *proportional* to *n*.
- $f_B(n) = n^2 + 1$ is *order of* n^2, or $O(n^2)$.
  - It is (at most) roughly *proportional* to n^2.
- Any function whose *exact* (tightest) order is $O(n^2)$ is faster-growing than any $O(n)$ function.

- For large numbers of user records, the order n^2 function will always take more time.

BIG-O NOTATION

The growth of functions is often described using a special notation

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

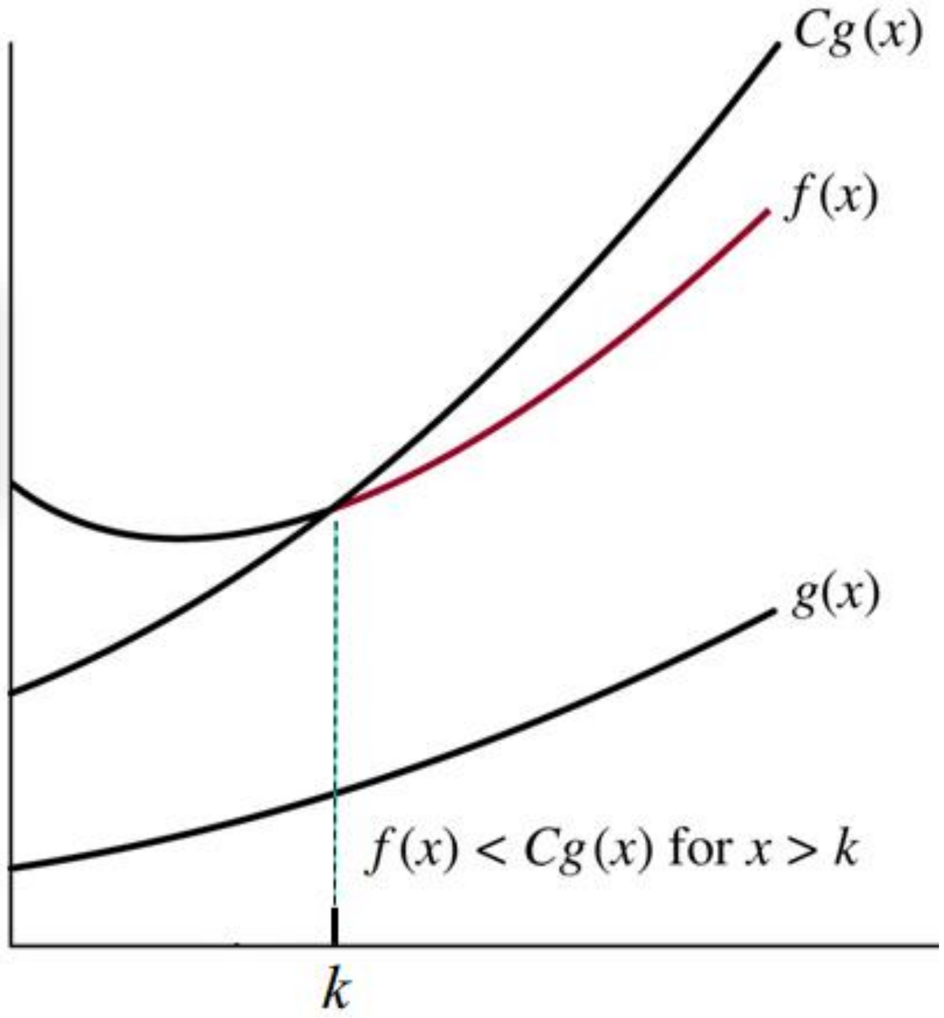
$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-oh of $g(x)$."]

Remark: Intuitively, the definition that $f(x)$ is $O(g(x))$ says that $f(x)$ grows slower that some fixed multiple of $g(x)$ as x grows without bound.

BIG -O NOTATION ILLUSTRATION

The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.



BIG -O NOTATION EXAMPLE

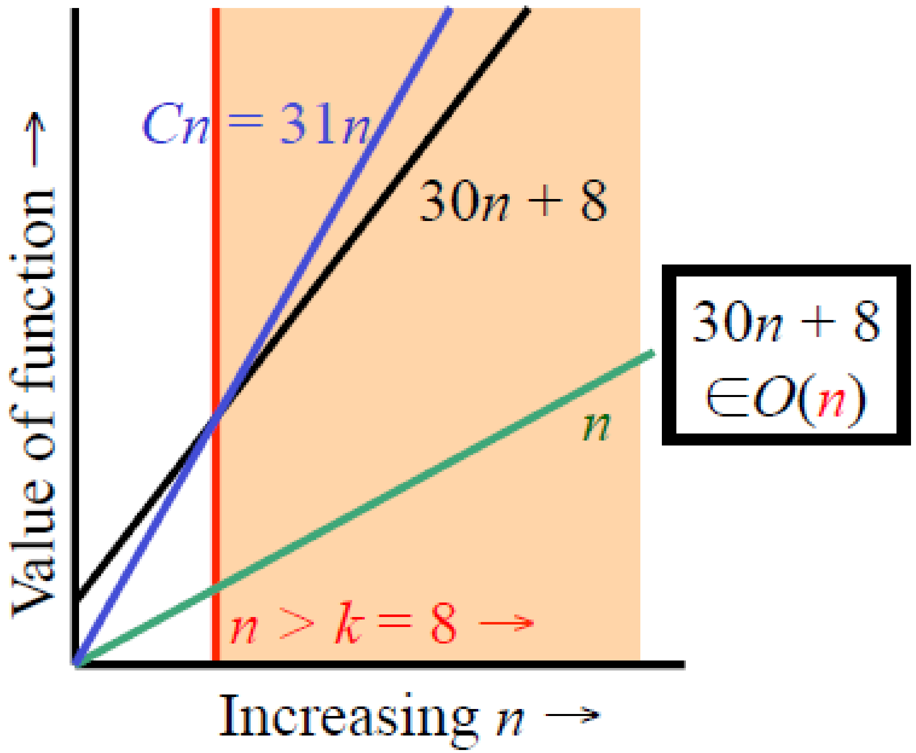
- Show that $30n + 8$ is $O(n)$.
  - Show $\exists C,k$ such that $\forall n > k$, $30n + 8 \leq Cn$.
    - Let $k = 8$. Assume $n > 8$ $(= k)$.

      Then, $30n + 8 < 30n + n = 31n$.

      Therefore, we can take $C = 31$ and $k = 8$ to show that $30n + 8$ is $O(n)$.

BIG -O NOTATION EXAMPLE

- Note $30n + 8$ isn't less than n *anywhere* ($n > 0$).
- It isn't even less than $31n$ *everywhere*.
- But it *is* less than $31n$ <u>everywhere to the right of $n = 8$</u>.



BIG -O NOTATION EXAMPLE

- $1 + 2 + \cdots + n \leq \overbrace{n + n + \cdots + n + n}^{n \text{ terms}} = n^2$.

  It follows that $1 + 2 + \cdots + n$ is $O(n^2)$, taking $C = 1$ and $k = 1$ as witnesses.

$$\text{Note: } 1 + 2 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$= \frac{1}{2}n^2 + \frac{1}{2}n$$

BIG-Ω NOTATION

- Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers.
- $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that $|f(x)| \geq C|g(x)|$ whenever $x > k$.
- This is read as "$f(x)$ is big-Omega of $g(x)$."
- $f(x)$ is $\Omega(g(x))$ if and only if $g(x)$ is $O(f(x))$

BIG-Ω NOTATION

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that

$$|f(x)| \geq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-Omega of $g(x)$."]

BIG-Θ NOTATION

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$. When $f(x)$ is $\Theta(g(x))$ we say that f is big-Theta of $g(x)$, that $f(x)$ is of *order* $g(x)$, and that $f(x)$ and $g(x)$ are of the *same order*.

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

GROWTH RATE FUNCTIONS

$O(1)$ Time requirement is **constant**, and it is independent of the problem's size.

$O(\log_2 n)$ Time requirement for a **logarithmic** algorithm increases increases slowly as the problem size increases.

$O(n)$ Time requirement for a **linear** algorithm increases directly with the size of the problem.

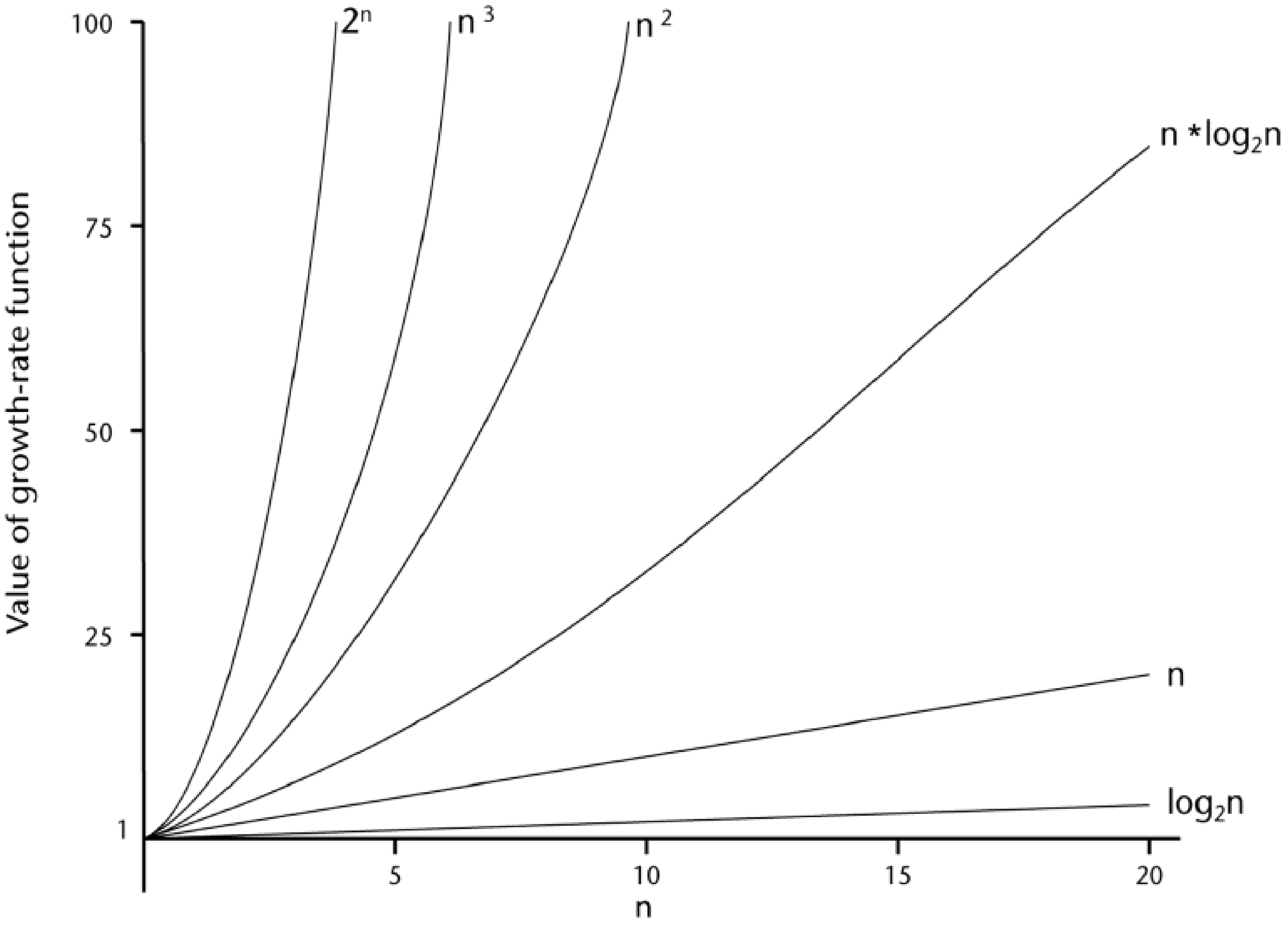
$O(n*\log_2 n)$ Time requirement for a **$n*\log_2 n$** algorithm increases more rapidly than a linear algorithm.

$O(n^2)$ Time requirement for a **quadratic** algorithm increases rapidly with the size of the problem.

$O(n^3)$ Time requirement for a **cubic** algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm.

$O(2^n)$ As the size of the problem increases, the time requirement for an **exponential** algorithm increases too rapidly to be practical.

GROWTH RATE FUNCTIONS VISUAL REPRESENTATION



ANALYSIS OF ALGORITHM

- When we analyze algorithms, we should employ mathematical techniques that analyze algorithms independently of *specific implementations, computers, or data.*

- To analyze algorithms:
  - First, we start to count the number of significant operations in a particular solution to assess its efficiency.
  - Then, we will express the efficiency of algorithms using growth functions.

THE EXECUTION TIME OF ALGORITHMS

- Each operation in an algorithm (or a program) has a cost.

  ➔ Each operation takes a certain of time.

```
count = count + 1;
```
➔ take a certain amount of time, but it is constant

A sequence of operations:

`count = count + 1;`	Cost: c_1
`sum = sum + count;`	Cost: c_2

➔ Total Cost $= c_1 + c_2$

GROWTH-RATE FUNCTIONS

	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    i = i + 1;`	c4	n
`    sum = sum + i;`	c5	n
`}`		

$$
\begin{aligned}
T(n) &= c1 + c2 + (n+1)*c3 + n*c4 + n*c5 \\
&= (c3+c4+c5)*n + (c1+c2+c3) \\
&= a*n + b
\end{aligned}
$$

➔ So, the growth-rate function for this algorithm is **O(n)**

GROWTH-RATE FUNCTIONS

- **O(1)**
- Big O notation O(1) represents the complexity of an algorithm that always execute in same time or space regardless of the input data.

```
void printFirstElementOfArray(int arr[])
{
    printf("First element of array = %d",arr[0]);
}
```

- This function runs in O(1) time (or "constant time") relative to its input. The input array could be 1 item or 1,000 items, but this function would still just require one step.

GROWTH-RATE FUNCTIONS

- **O(n) :** Big O notation O(N) represents the complexity of an algorithm, whose performance will grow linearly (in direct proportion) to the size of the input data.
- **O(n) example :** The execution time will depend on the size of array. When the size of the array increases, the execution time will also increase in the same proportion (linearly)

This function runs in O(n) time (or "linear time"), where n is the number of items in the array. If the array has 10 items, we have to print 10 times. If it has 1000 items, we have to print 1000 times.

```
void printAllElementOfArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
    }
}
```

GROWTH-RATE FUNCTIONS

- **O(n^2) example**

Here we're nesting two loops. If our array has n items, our outer loop runs n times and our inner loop runs n times for each iteration of the outer loop, giving us n^2 total prints

Thus this function runs in O(n^2) time (or "quadratic time"). If the array has 10 items, we have to print 100 times. If it has 1000 items, we have to print 1000000 times.

Other examples: Bubble sort

```
void printAllPossibleOrderedPairs(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
        }
    }
}
```

PROPERTIES OF GROWTH-RATE FUNCTIONS

1. *We can ignore low-order terms in an algorithm's growth-rate function.*
   - If an algorithm is $O(n^3+4n^2+3n)$, it is also $O(n^3)$.
   - We only use the higher-order term as algorithm's growth-rate function.

2. *We can ignore a multiplicative constant in the higher-order term of an algorithm's growth-rate function.*
   - If an algorithm is $O(5n^3)$, it is also $O(n^3)$.

3. $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
   - We can combine growth-rate functions.
   - If an algorithm is $O(n^3) + O(4n)$, it is also $O(n^3 +4n^2)$ ➔ So, it is $O(n^3)$.
   - Similar rules hold for multiplication.

DROP THE CONSTANTS

- When you're calculating the big O complexity of something, you just throw out the constants. Like:

```
void printAllItemsTwice(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
    }

    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
    }
}
```

This is O(2n), which we just call O(n).

```
void printFirstItemThenFirstHalfThenSayHi100Times(int arr[], int size)
{
    printf("First element of array = %d\n",arr[0]);

    for (int i = 0; i < size/2; i++)
    {
        printf("%d\n", arr[i]);
    }

    for (int i = 0; i < 100; i++)
    {
        printf("Hi\n");
    }
}
```

This is O(1 + n/2 + 100), which we just call O(n).

DROP THE LESS SIGNIFICANT TERMS

Here our runtime is $O(n + n^2)$, which we just call $O(n^2)$.

Similarly:

$O(n^3 + 50n^2 + 10000)$ is $O(n^3)$
$O((n + 30) * (n + 5))$ is $O(n^2)$
Again, we can get away with this because the less significant terms quickly become, well, less significant as n gets big.

```c
void printAllNumbersThenAllPairSums(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
    }

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d\n", arr[i] + arr[j]);
        }
    }
}
```

Remember,

for big O notation we're looking at what happens as **n** gets arbitrarily large. As **n** gets really big, adding 100 or dividing by 2 has a decreasingly significant effect.

WHAT TO ANALYZE

- An algorithm can require different times to solve different problems of the same size.
  - Eg. Searching an item in a list of n elements using sequential search. → Cost: 1,2,...,n
- ***Worst-Case Analysis*** –The maximum amount of time that an algorithm require to solve a problem of size n.
  - This gives an upper bound for the time complexity of an algorithm.
  - Normally, we try to find worst-case behavior of an algorithm.
- ***Best-Case Analysis*** –The minimum amount of time that an algorithm require to solve a problem of size n.
  - The best case behavior of an algorithm is NOT so useful.
- ***Average-Case Analysis*** –The average amount of time that an algorithm require to solve a problem of size n.
  - Sometimes, it is difficult to find the average-case behavior of an algorithm.
  - We have to look at all possible data organizations of a given size n, and their distribution probabilities of these organizations.
  - ***Worst-case analysis is more common than average-case analysis.***

LINEAR SEARCH ALGORITHM

If Key = 41

Then No of comparisons will be : 6

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

Best Case : If the key element is present at the 1st index then it is the best case

Best Case Time = 1 (as it will take constant time)

Complexity : O(1)

LINEAR SEARCH ALGORITHM

- Worst Case : Searching a key at last index (i.e. 52)
- Worst Case Time : O(n)

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	$\boxed{52}$

- Average Case : All possible case time / no of cases

$1+2+3+4+5+6+7+8+9 \ / \ 9$

$1+2+3+\dots\dots..+n/n$

$(\cancel{n}(n+1) \ /2 \,) \ / \ \cancel{n}$

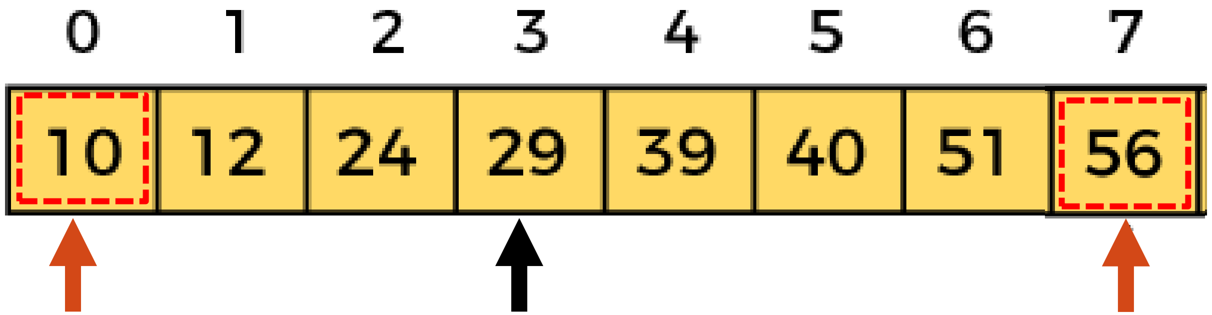
$(n+1) \ / \ 2$

$O(n)$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

BINARY SEARCH

Minimum No. of Comparison –> Best Case
Maximum No of Comparison -> Worst Case



Array Size N = 8

BEST CASE :
Search Element is equal to the very first middle element.

BEST CASE TIME = O(1)

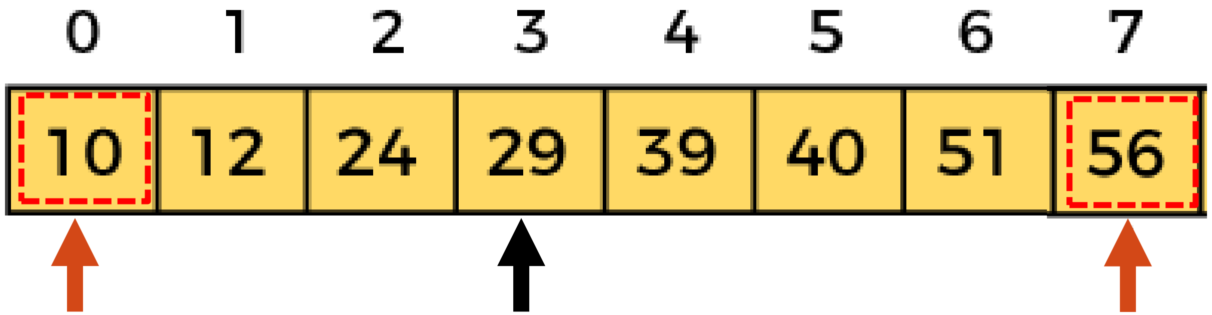
Search Element = 29

m =(L + R) / 2
m = 0 + 7 / 2
m = 3

Middle element = Search Element

BINARY SEARCH

- Array Size N = 8



- WORST CASE :
- Search Element is present at either beginning of the array or end of the array.

Search Element = 10

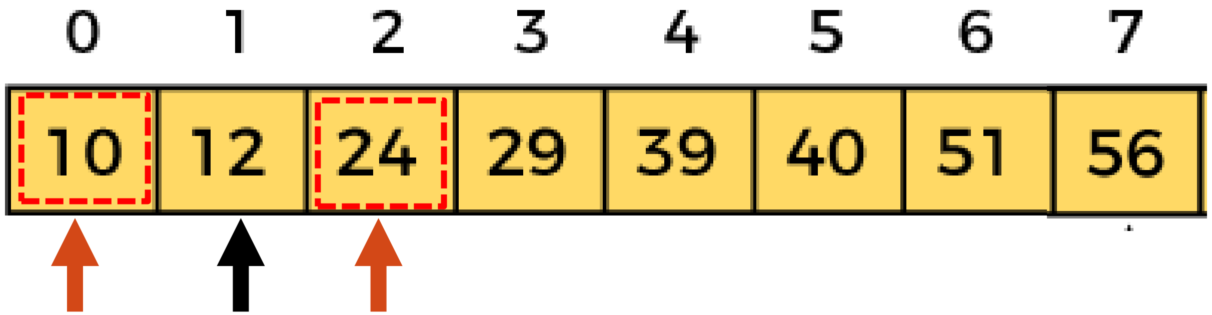
1^{st} Comparison
m =(L + R) / 2
m = 0 + 7 / 2
m = 3

Middle element ≠ Search Element

BINARY SEARCH

- Array Size N = 8

- WORST CASE :
- Search Element is present at either beginning of the array or end of the array.



Search Element = 10
2nd Comparison

m =(L + R) / 2
m = 0 + 2 / 2
m = 1

Middle element ≠ Search Element

BINARY SEARCH

- Array Size N = 8

0	1	2	3	4	5	6	7
10	12	24	29	39	40	51	56

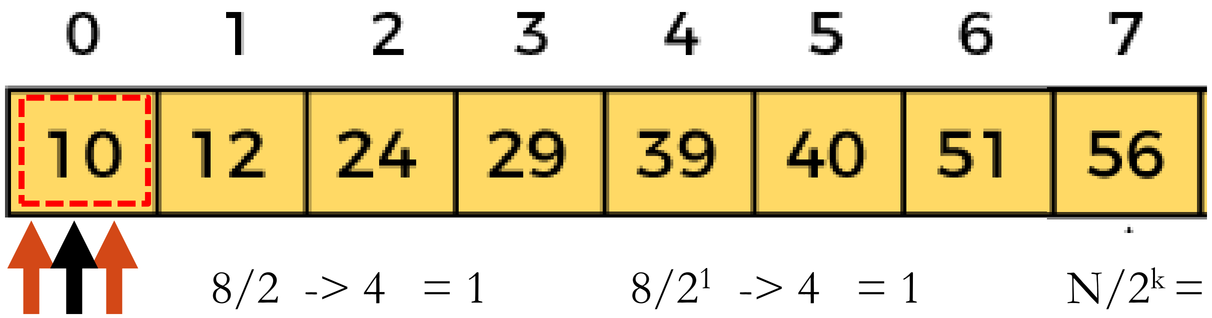
- WORST CASE :
- Search Element is present at either beginning of the array or end of the array.

Search Element = 10
3rd Comparison

m =(L + R) / 2
m = 0 + 0 / 2
m = 0
Middle element = Search Element

BINARY SEARCH

- Array Size N = 8



- WORST CASE :
- Search Element is present at either beginning of the array or end of the array.
- O (log_2 n)

8/2 -> 4 = 1
4/2 -> 2 = 2
2/2 -> 1 =3

$8/2^1$ -> 4 = 1
$8/2^2$ -> 2 = 2
$8/2^3$ -> 1 =3

$N/2^k = 1$
$N = 2^k$
$log_2 N = k \, log_2 2$
$k = log_2 n$

0	1	2	3	4	5	6	7	← an array with size 8
3	2	3	1	3	2	3	4	← # of iterations

The average # of iterations = 21/8 < $\log_2 8$