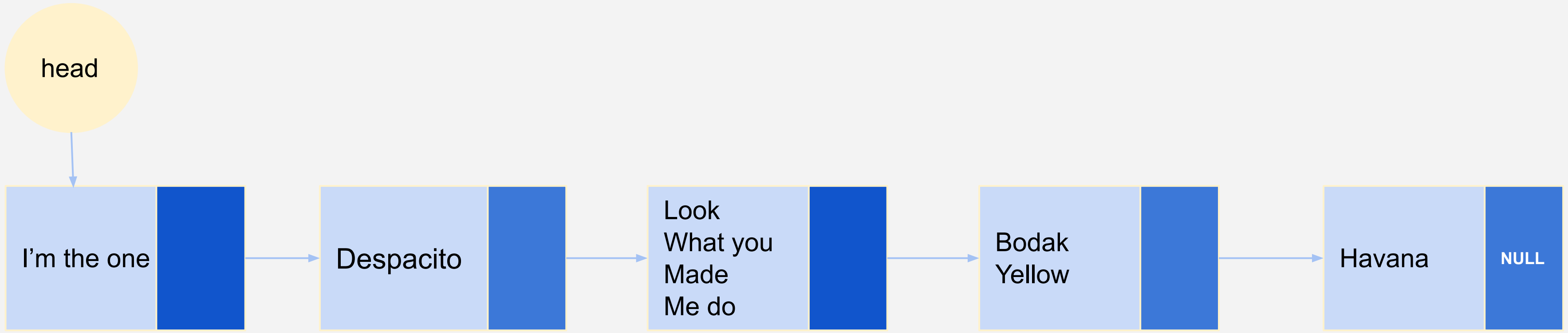
CSCA48 – Week 8

Efficiency of algorithms

Gawde, Summer 2022

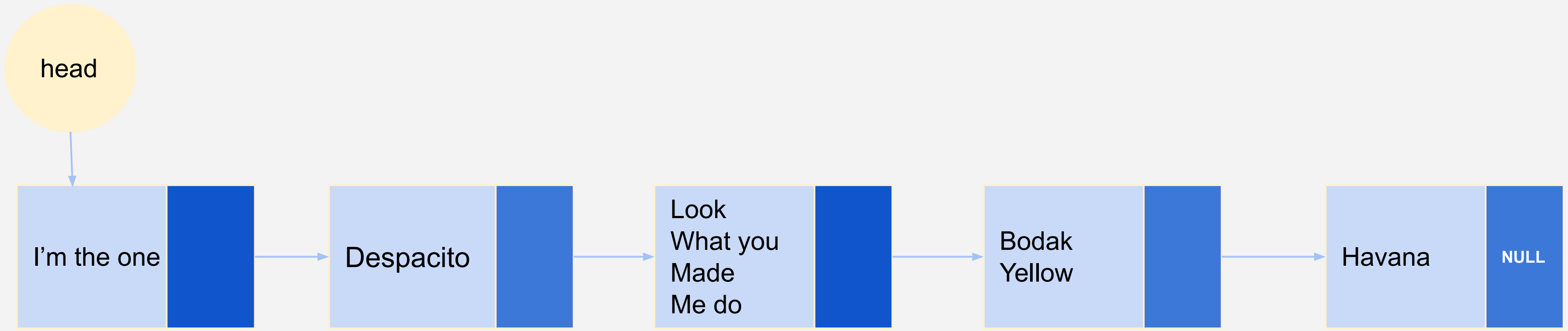
Linked List Issues

- List traversal for **searching** the list
    - Longer wait time to find a node
    - If linked list contains database
        - Cost of answering a query becomes too large



Search time: Best, Worst, Average

- Best case: item found in the first node
- Worst case: item found in the tail of the list
- Average case: item found somewhere in the middle



Is it just a wrong data structure??

- Maybe list is not a good idea
- Maybe we should work with arrays?

I’m the one	Despacito	Look what you made me do	Bodak Yellow	Havana

- Observations:
  - Items are randomly ordered
  - In order to look for an item we need to start at the beginning and go through the entire array
    - Also known as **linear search**
  - Best case, worst case, average case?
    - Same??
  - Did we solve the problem??
  - Conclusion:
    - So it’s not just the data structure
    - We need to **organize our data** better

How to organize data?

- Just like a dictionary or a phone book: *alphabetically*
- Order the data based on the some *key* in data
- Key
  - Some unique value in the data field that represents the node
- We get sorted array
- How does sorted array help us??
  - Now we can estimate the approximate location of the item we are looking for

Bodak Yellow	Despacito	Havana	In my feelings	Look what you made me do

Binary Search

- Approach:
    - Step 1: Find the middle
    - Step 2: If it's the item we are looking for..
        - Done!!
    - Else
    - Step 3: If item is less than middle, look for the item in first half by repeating steps 1 and 2
    - Step 4: If item is greater than middle, look for the item in second half by repeating steps 1 and 2
- Suppose we are looking for song "In my feelings"

Bodak Yellow
Despacito
Havana
In my feelings
Look what you made me do

Binary Search

- Suppose we are looking for song “In my feelings”

- Middle of the array: Havana
- Havana < In my feelings

Bodak Yellow
Despacito
Havana
In my feelings
Look what you made me do

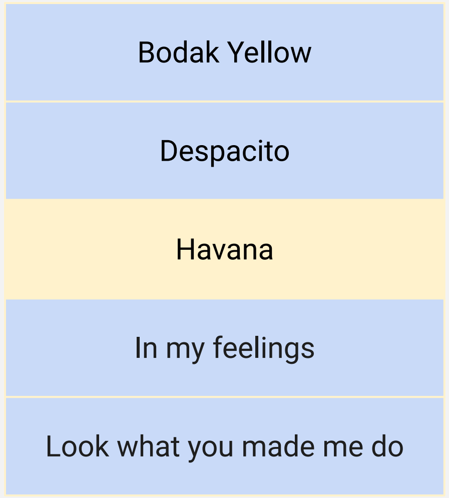
- Middle of the array: In my feelings
- Done!!

In my feelings
Look what you made me do

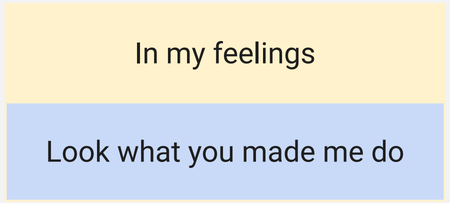
Binary Search

- We looked at just 2 items in the array

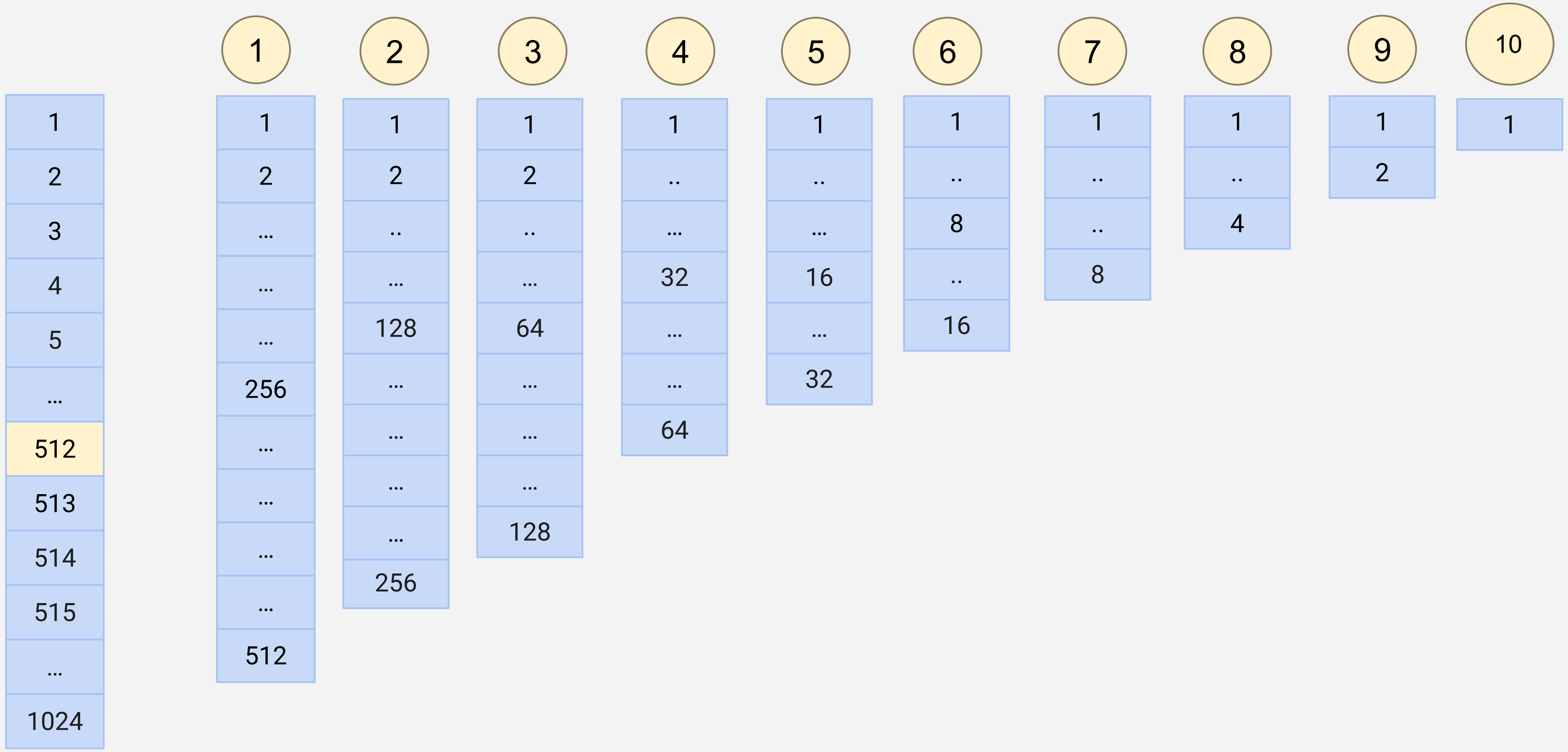
- Middle of the array: Havana
- Havana < In my feelings



- Middle of the array: In my feelings
- Done!!



Binary search for 1024 entries



How many items in general?? (for binary search)

```
int binarySearch(int arr[], int lf, int rh, int s)
{
  while (rh >= lf)
   {
     int mid = lf + (rh - lf) / 2;

     if (arr[mid] == s)
        return mid;

     if (arr[mid] < s)
        lf = mid + 1;

     else
        rh = mid - 1;
    }

    return -1;
}
```

Iterations	Items left(N)	Expressed different way
Iter 0		
Iter 1		
Iter 2		
Iter 3		
…		

- Questions we want to answer:
  - When do you stop (in the worst case)??
  - How many items remain in the last iteration??
  - How many iterations??
  - Can we generalize this??

How many items in general?? (for binary search)

```
int binarySearch(int arr[], int lf, int rh, int s)
{
  while (rh >= lf)
   {
     int mid = lf + (rh - lf) / 2;

     if (arr[mid] == s)
        return mid;

     if (arr[mid] < s)
        lf = mid + 1;

     else
        rh = mid - 1;
    }

    return -1;
}
```

Iterations	Items left(N)	Expressed different way
Iter 0	1024	
Iter 1	512	
Iter 2	256	
Iter 3	128	
…	…	

- Questions we want to answer:
  - When do you stop (in the worst case)??
  - How many items remain in the last iteration??
  - How many iterations??
  - Can we generalize this??

How many items in general?? (for binary search)

```
int binarySearch(int arr[], int lf, int rh, int s)
{
  while (rh >= lf)
   {
     int mid = lf + (rh - lf) / 2;

     if (arr[mid] == s)
        return mid;

     if (arr[mid] < s)
        lf = mid + 1;

     else
        rh = mid - 1;
    }

      return -1;
}
```

Iterations	Items left	Expressed in power of 2
Iter 0	1024	
Iter 1	512	
Iter 2	256	
Iter 3	128	
…	…	
Iter k	**1**	

How many items in general?? (for binary search)

```
int binarySearch(int arr[], int lf, int rh, int s)
{
  while (rh >= lf)
   {
     int mid = lf + (rh - lf) / 2;

     if (arr[mid] == s)
        return mid;

     if (arr[mid] < s)
        lf = mid + 1;

     else
        rh = mid - 1;
    }

      return -1;
}
```

Iterations	Items left	Expressed in power of 2
Iter 0	1024	$1024/2^0$
Iter 1	512	$1024/2^1$
Iter 2	256	$1024/2^2$
Iter 3	128	$1024/2^3$
…	…	…
Iter k	**1**	$1024/2^k$

- We want to find the value of k
- Input size = N = 1024
- Replace 1024 with N

How many items in general?? (for binary search)

```
int binarySearch(int arr[], int lf, int rh, int s)
{
  while (rh >= lf)
   {
     int mid = lf + (rh - lf) / 2;

     if (arr[mid] == s)
        return mid;

     if (arr[mid] < s)
        lf = mid + 1;

     else
        rh = mid - 1;
    }

      return -1;
}
```

Iterations	Items left	Expressed in power of 2
Iter 0	1024	$N/2^0$
Iter 1	512	$N/2^1$
Iter 2	256	$N/2^2$
Iter 3	128	$N/2^3$
…	…	…
Iter k	**1**	$N/2^k$

- We want to find the value of k
- Input size = N = 1024

  $1 = N/2^k$

  Solving for k,

  $N = 2^k$

  Taking log on both sides: $k = \log_2 N$

How many items in general?? (for binary search)

```
int binarySearch(int arr[], int lf, int rh, int s)
{
  while (rh >= lf)
   {
     int mid = lf + (rh - lf) / 2;

     if (arr[mid] == s)
        return mid;

     if (arr[mid] < s)
        lf = mid + 1;

     else
        rh = mid - 1;
    }

      return -1;
}
```

Iterations	Items left	Expressed in power of 2
Iter 0	1024	$N/2^0$
Iter 1	512	$N/2^1$
Iter 2	256	$N/2^2$
Iter 3	128	$N/2^3$
…	…	…
Iter k	**1**	$N/2^k$

Answer is **$k = \log_2(N)$**

Our example:

$k = \log_2(1024)$

$k = 10$

Efficiency so far

N	Binary Search: worst ($\log_2 n$)	Linear search: avg (N/2)	Linear search: worst (N)
2	1	1	2
4	2	2	4
8	3	4	8
16	4	8	16
..			..
1024	10	512	1024
2048	11	1024	2048
4096	12	2048	4096
..			..
33554332	25	16777216	33554332

Computational complexity

- How algorithms perform for increasingly larger data collection (Increasing order of **N**)
- Linear search (worst case)
  - Examines **N** items
- Binary search (worst case)
  - Examines **$\log_2(N)$** items
- In terms of functions:
  - $f_{binary\ search} = log_2(N)$
  - $f_{linear\ search} = N$

Visualization of complexity

Linear search 🟥

Binary search 🟦

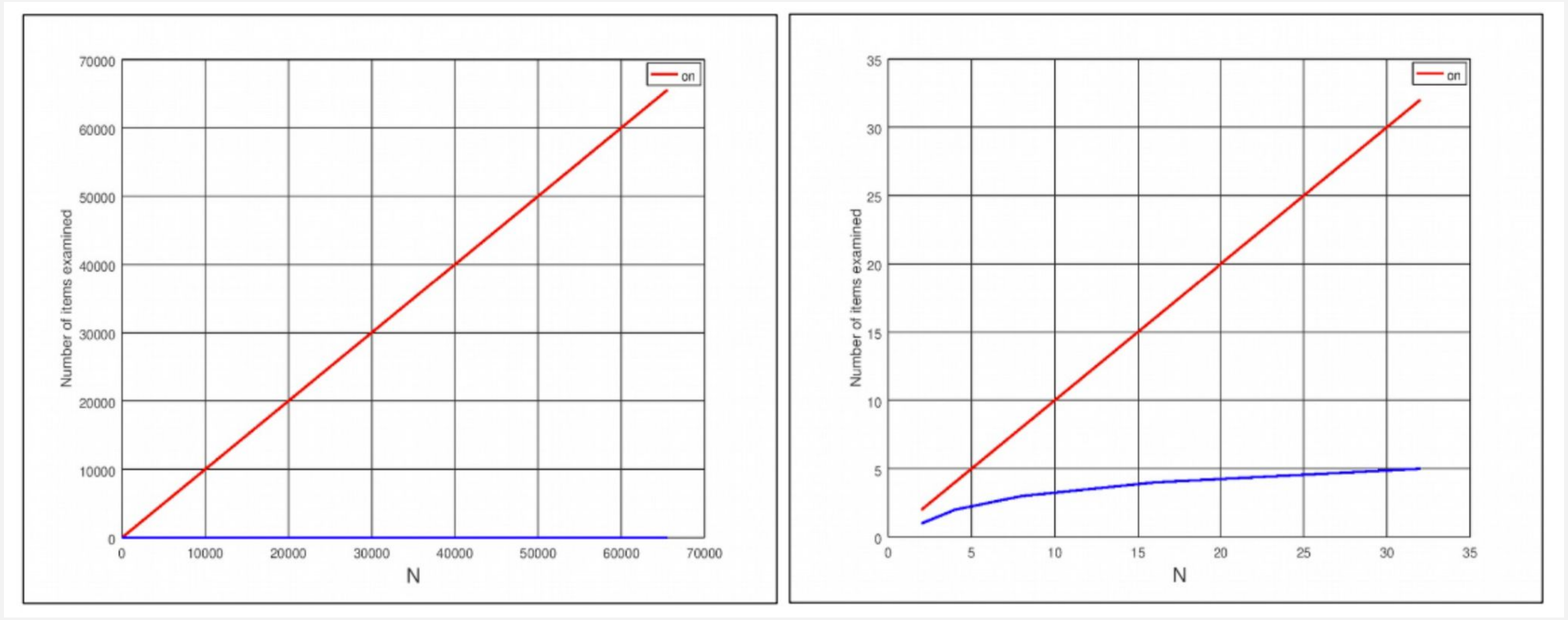


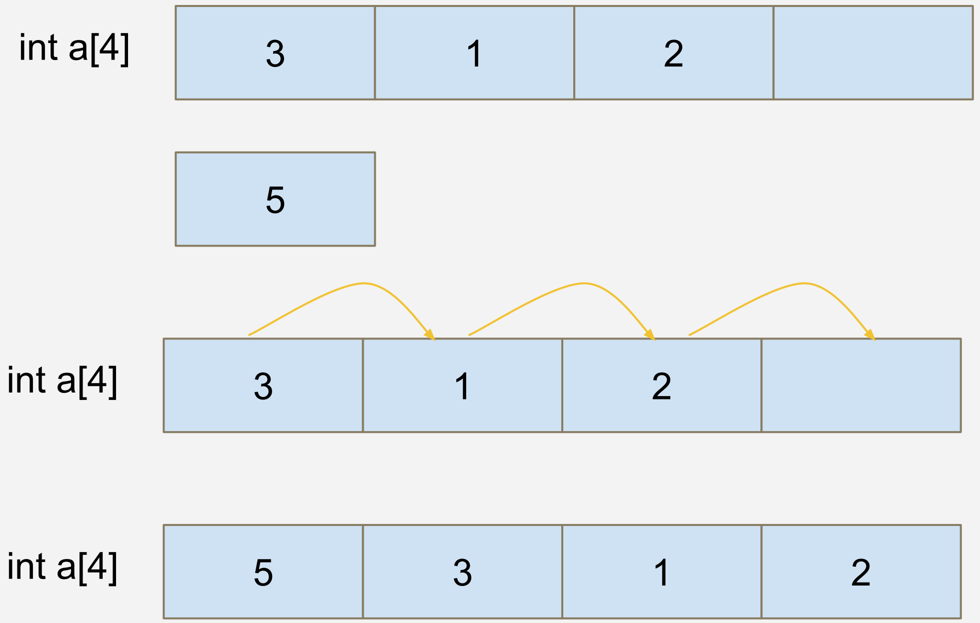
Image taken from Unit4 Notes© F.Estrada, M.Ahmadzadeh, B.Harrington 2020

How to measure the computational complexity?

- Computational complexity can be measured in terms of **time** and space
- Measure the running time
    - Approach 1(exact run time):
        - Exact time to run a program that uses a data structure on different machine perhaps in different programming languages
        - Use timer to note the exact time
        - Repeat the process for different size of inputs(N)
        - Not practical
    - Approach 2(measure growth rate):
        - Approximate running time of data structure for different size of input(N)
            - Find out **growth rate**
            - Discard less contributing factors

Computational complexity example

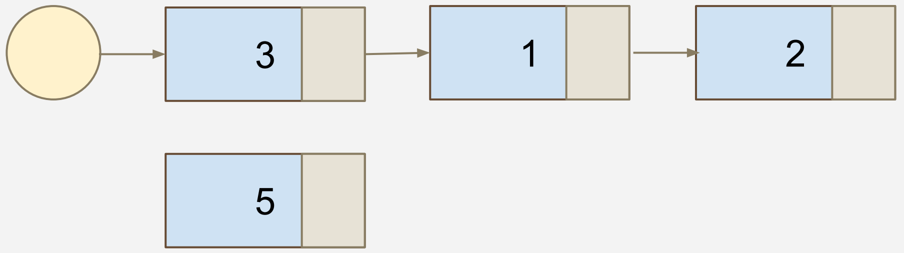
Add an element at the beginning of data structure. (Arrays vs Linked lists)
Arrays:



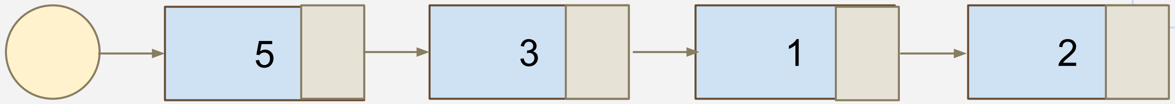
- For an array of 3 elements
  - Each shift takes **1 unit** of time
  - 3 shifts: 3 units
  - **Total time = 3**
- For an array of 10000 elements?
  - **Total time??**

Computational complexity example

Add an element at the beginning of data structure.
Linked list:



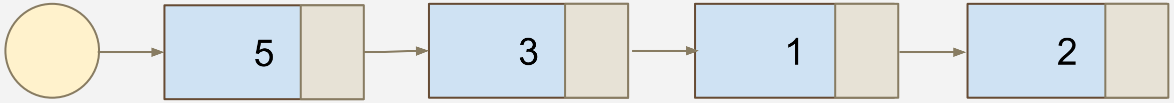
- For a linked list of 3 elements
  - Adjusting head pointer takes 1 unit of time
  - Linking new node takes 1 unit of time
  - **Total time = 2**
- For an array of 10000 elements?
  - **Total time??**



Computational complexity example

Add an element at the beginning of data structure.

Which data structure would you choose??



- What factors did you consider while making that decision?
    - As the size of data structure grows, linked list would take just 1 unit to add an element at the head
        - Dominating operation (insertion of data at the beginning of data structure) in your application
        - Behaviour of data structure as size grows(N) in terms of computational complexity

How to calculate computational complexity (Approach 1)

- Coming back to our search example:
- Suppose we ran the linear search algorithm on different machines and got following output:

For input size N	Machine 1	Machine 2	Machine 3
Linear search on arrays	.75 □ N	1.25 □ N	2.43 □ N
Binary search on arrays	1.15 □ $\log_2(N)$	1.75 □ $\log_2(N)$	15.245 □ $\log_2(N)$

- Run-time for Linear search on arrays:
  - *f(n) = constant □ n = function of n*
- Run-time for Binary search on arrays:
  - *f(n) = constant □ $\log_2(n)$ = function of $\log_2(n)$*

Big O notation

- Measures the performance of an algorithm by providing the **order of growth** of a function
- It gives the **least upper bound** on function and makes sure the function doesn't grow faster than this upper bound
- *$f(n)$* denotes the run-time of algorithm
- *$O(g(n))$* denotes the least upper bound on a function

$$f(n) = O(g(n))$$

If and only if there exists some positive constant ***c*** such that for sufficiently large value of ***n***,

$$|f(n)| \leq c.g(n) \text{ for all } n \geq n_0$$

Big O notation

$f(n) = O(g(n))$

If and only if there exists some positive constant ***c*** such that for sufficiently large value of ***n***,

$|f(n)| \le c.g(n)$ for all $n \ge n_0$

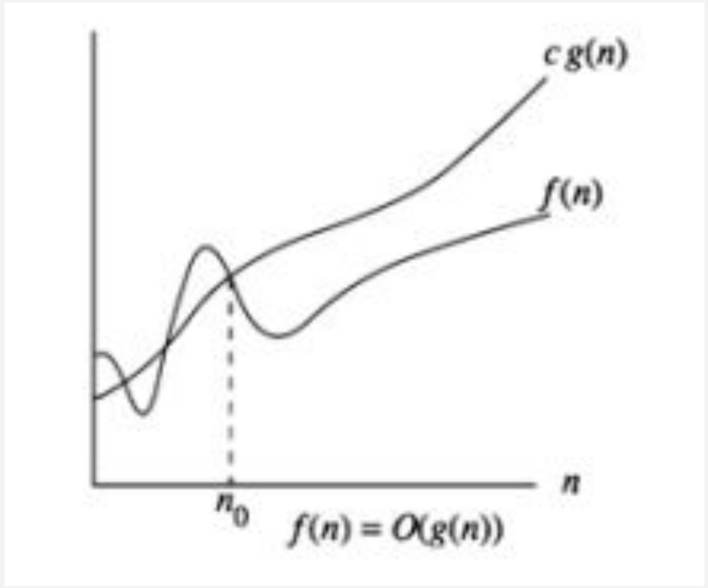


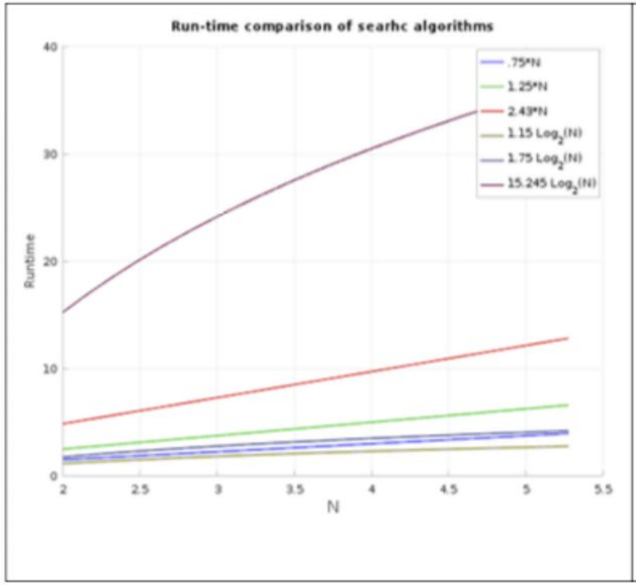
Image: Introduction to Algorithms, Third Edition | The MIT Press

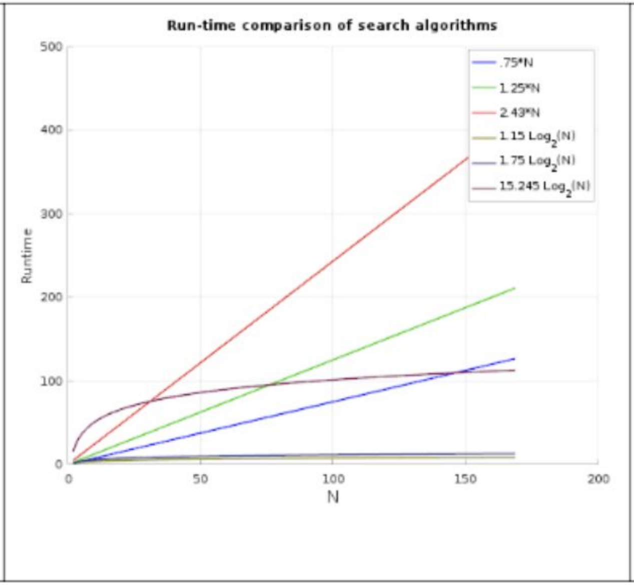
Complexity of linear and binary search

For input size N	Machine 1	Machine 2	Machine 3
Linear search on arrays	.75 □ N	1.25 □ N	2.43 □ N
Binary search on arrays	1.15 □ log$_2$(N)	1.75 □ log$_2$(N)	15.245 □ log$_2$(N)

- Run-time for Linear search on arrays:
    - ***O(n)*** the complexity of linear search is of order n
- Run-time for Binary search on arrays:
    - ***O(log$_2$(n))*** the complexity of binary search is of order log$_2$(n)

Visualizing run times





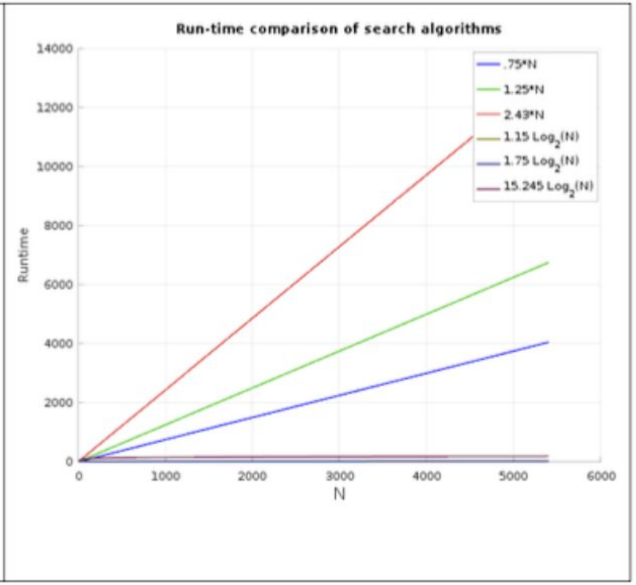


Image taken from Unit4 Notes© F.Estrada, M.Ahmadzadeh, B.Harrington 2020

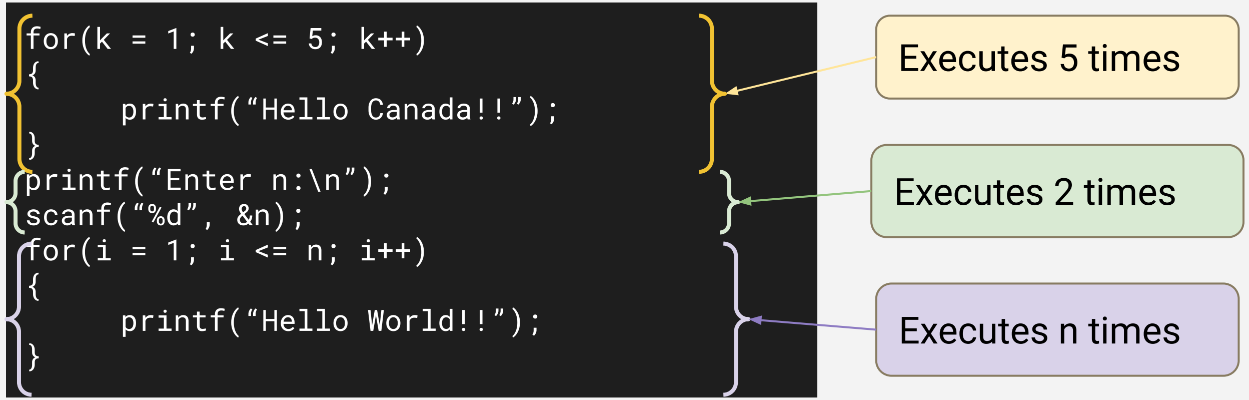
Example 1: Complexity of following program

```
for(i = 1; i <= n; i++)
{
     printf(“Hello World!!”);
}
```

Iterations	Value of i
Iter 1	i= 1
Iter 2	i = 2
Iter 3	i = 3
Iter 4	i = 4
Iter k	i = n

- Program has just 1 loop
- Loop repeats **k** times
- Value of k??
  - k = n;
  - What is the complexity??
  - f(n) = O(n)

Example 2: Complexity of following program



```
for(k = 1; k <= 5; k++)
{
      printf("Hello Canada!!");
}
printf("Enter n:\n");
scanf("%d", &n);
for(i = 1; i <= n; i++)
{
      printf("Hello World!!");
}
```

- Program has two for loops and 2 statements
  - Loop1: repeats 5 times (5 units)
  - Loop2: repeats **k** times (k = n units)
  - 2 statements (2 units)
- Total time: 5 + n + 2 = n + 7
- Complexity: f(n) = O(n)
- What about 7??
  - Ignore it..

Example 2: contd..

```
for(k = 1; k <= 5; k++)
{
     printf("Hello Canada!!");
}

printf("Enter n:\n");
scanf("%d", &n);

for(i = 1; i <= n; i++)
{
     printf("Hello World!!");
}
```

Total time: n + 7

Contributions of n vs contribution of 7 as n grows

Value of n	Total time	% n	% 7
n = 2	9		
n = 3	10		
n = 4	11		
n = 5	12		
n = 6	13		
n = 7	14		
n = 10	17		
n = 1000	1007		
n = 1000000	1000007		

Example 2: contd..

```
for(k = 1; k <= 5; k++)
{
     printf(“Hello Canada!!”);
}

printf(“Enter n:\n”);
scanf(“%d”, &n);

for(i = 1; i <= n; i++)
{
     printf(“Hello World!!”);
}
```

Total time: n + 7

Contributions of n vs contribution of 7 as n grows

Value of n	Total time	% n	% 7
n = 2	9	22.22	77.77
n = 3	10	30	70
n = 4	11	36.36	63.63
n = 5	12	41.66	58.33
n = 6	13	46.15	53.85
n = 7	14	50	50
n = 10	17	58.82	41.18
n = 1000	1007	99.30	0.70
n = 1000000	1000007	99.99993	0.00007

Example 2: verify the upper bound with formula

```
for(k = 1; k <= 5; k++)
{
     printf("Hello Canada!!");
}

printf("Enter n:\n");
scanf("%d", &n);

for(i = 1; i <= n; i++)
{
     printf("Hello World!!");
}
```

If we find values of ***n_0*** and ***c***
We can prove ***$f(n) = O(g(n))$***

Total time: n + 7

$f(n) = O(g(n))$

If and only if there exists some positive constant ***c*** such that for sufficiently large value of ***n***,

$|f(n)| \le c.g(n)$ for all $n \ge n_0$

From Algorithm Complexity to Problem Complexity

- Goal: how efficient two different algorithms for finding a particular item in an array can be
- Two algorithms:
  - Linear search
  - Binary search
- Problem Complexity
  - We study the actual problem of finding an element
  - Theoretical lower-limit on how much work the best possible algorithm has to do to find an element

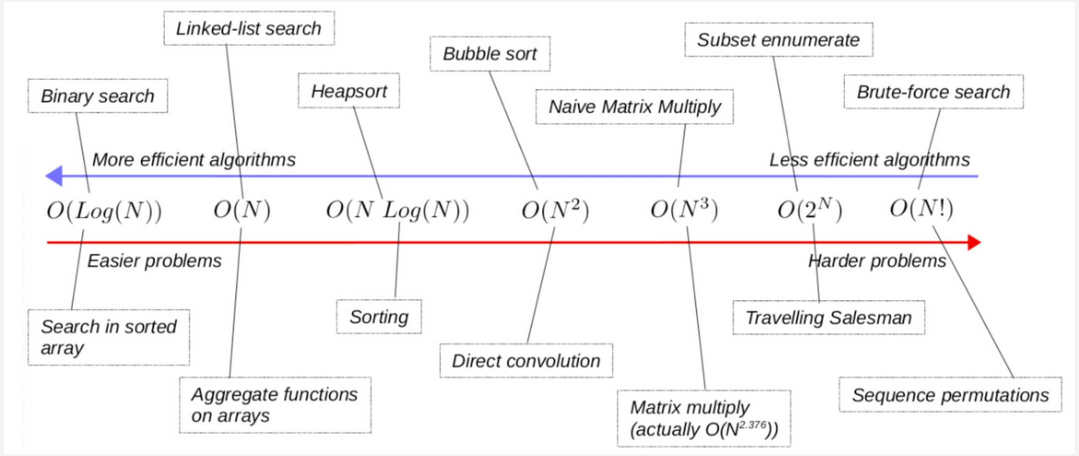


Image taken from Unit4 Notes© F.Estrada, M.Ahmadzadeh, B.Harrington 2020

Computational Complexity Measures

- Depending on the problem, computational complexity can be measured:
    - the number of times an item in a collection is accessed,
        - Searching through the list of items
    - the run-time of an algorithm
        - Number of instructions executed
    - the number of mathematical operations a certain function has to perform.

How to make searching for information more efficient

- Arrays or Linked Lists for searching??

- What do we need?
  - Cost of sorting + $O(\log_2 N)$

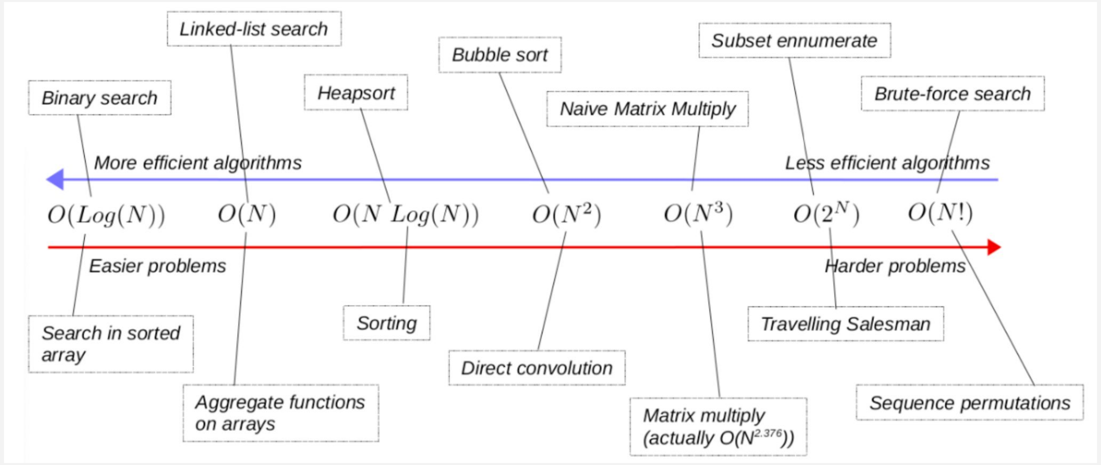


Image taken from Unit4 Notes© F.Estrada, M.Ahmadzadeh, B.Harrington 2020

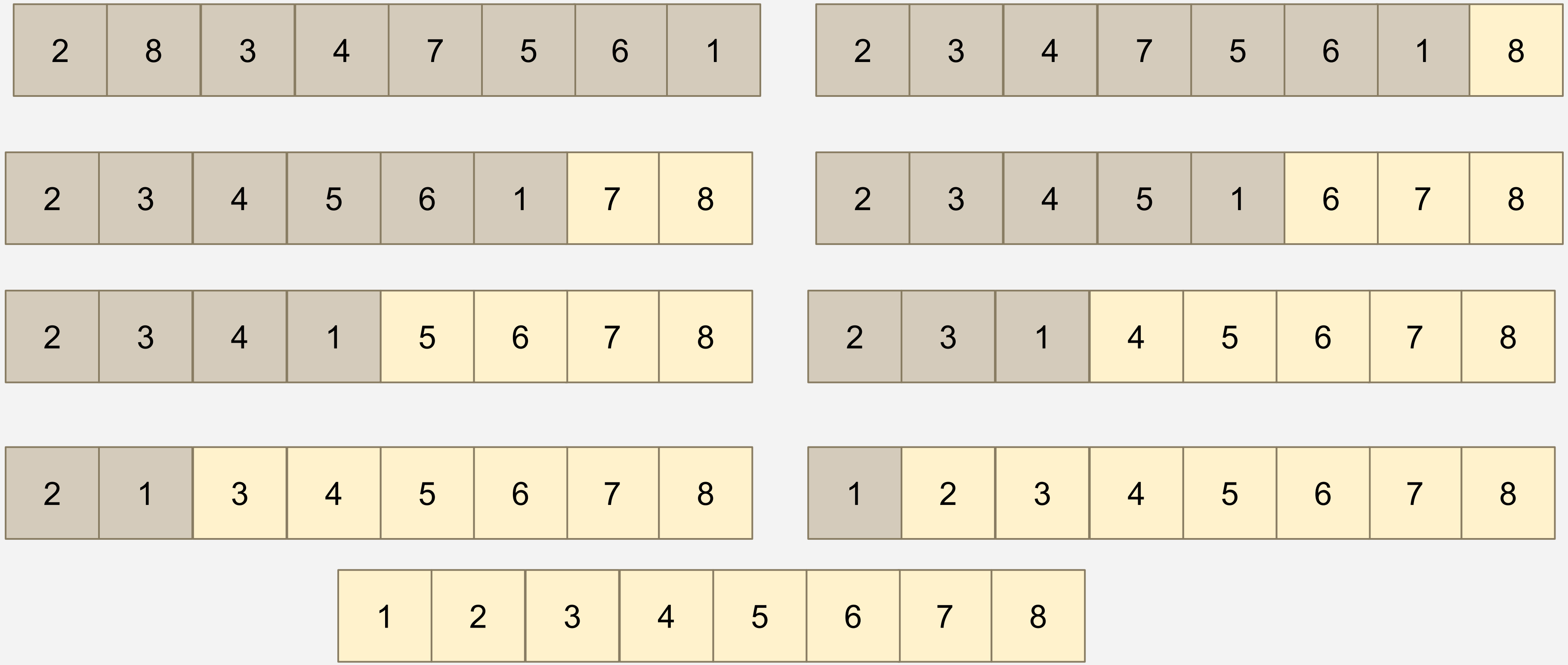
The Cost of Sorting - Bubble Sort

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

```
void BubbleSort(int array[], int N)
{
  // Traverse an array swapping any entries such that array[j] > array[j+1].
  // Keep doing that until the array is sorted (at most, N iterations)
        int t;
        for (int i=0; i<N; i++)
        {
                for(int j=0; j<N-1; j++)
                {
                        if(array[j] > array[j+1])
                        {
                                t = array[j+1];
                                array[j]=array[j+1];
                                array[j+1]=t;
                        }


                }
        }
}
```

Bubble Sort Example



Nested Loop Structure

Outer loop has N iterations

Inner loop has N-1 iterations

The inner loop updates at most 2 array entries

So the complexity of the function is $N * (N - 1) * 2$

$= 2(N^2 - N)$

$\rightarrow O(N^2)$.

How??

Example 2: verify the upper bound with formula

```
void BubbleSort(int array[], int N)
{
     int t;
       for (int i=0; i<N; i++)
       {
              for(int j=0; j<N-1; j++)
              {
                     if(array[j] > array[j+1])
                     {
                            t = array[j+1];
                            array[j]=array[j+1];
                            array[j+1]=t;
                     }
              }
       }
}
```

Total time: $2(n^2 - n)$

If we find values of n_0 and c
We can prove $f(n) = O(g(n))$

$f(n) = O(g(n))$

If and only if there exists some positive constant ***c*** such that for sufficiently large value of ***n***,

$|f(n)| \leq c.g(n)$ *for all* $n \geq n_0$

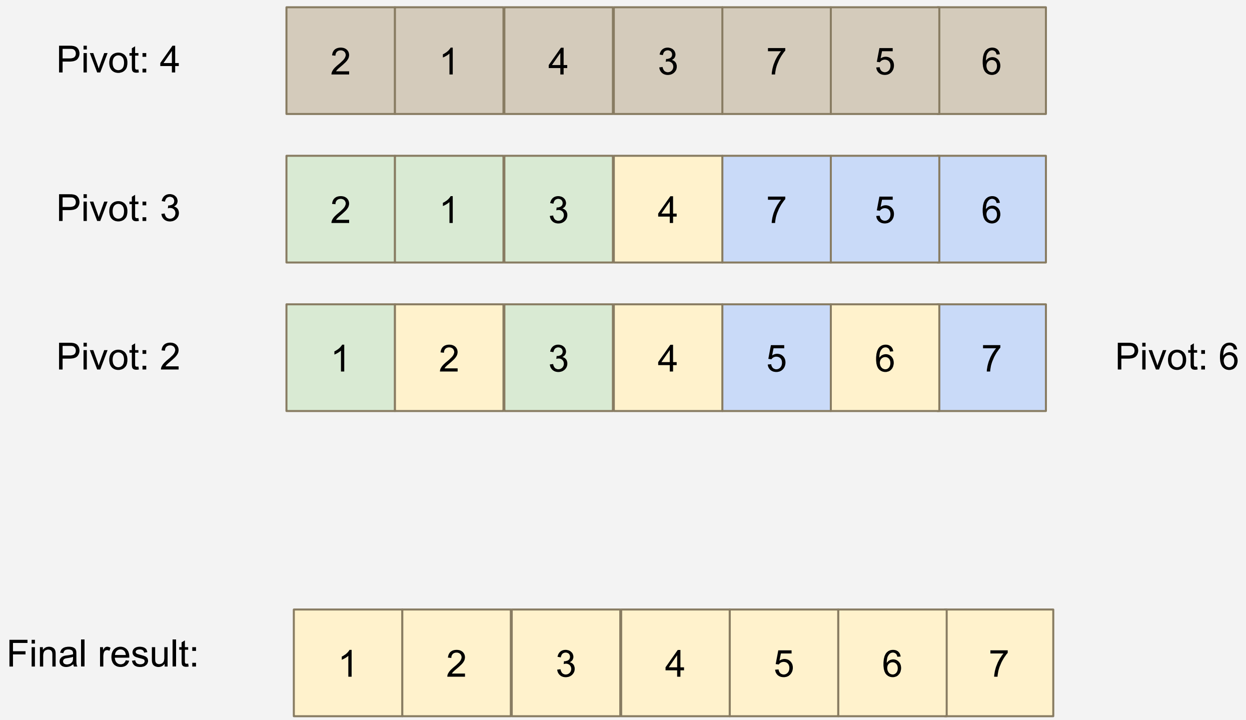
Bubble Sort + Binary Search

- Complexity
  - Sorting complexity
    - $O(n^2)$
  - Binary Search complexity
    - $O(\log_2 n)$
  - Total
    - $O(n^2) + O(\log_2 n)$
- Bubble Sort is not the best!
- At best, we can get $O(n \log n)$ for sorting..

Quicksort

- Idea of Quicksort
  - Partition array A into A[1..q − 1], A[q], and A[q + 1..n] such that
    - Each element in A[1..q − 1] is ≤ A[q].
    - Each element in A[q + 1..n] is > A[q].
  - The element A[q] is called **pivot**.
  - Recursively sort (in place) each subarray.
- Average complexity:
  - $O(n \log n)$
- Worst case complexity:
  - $O(n^2)$

Quicksort Example

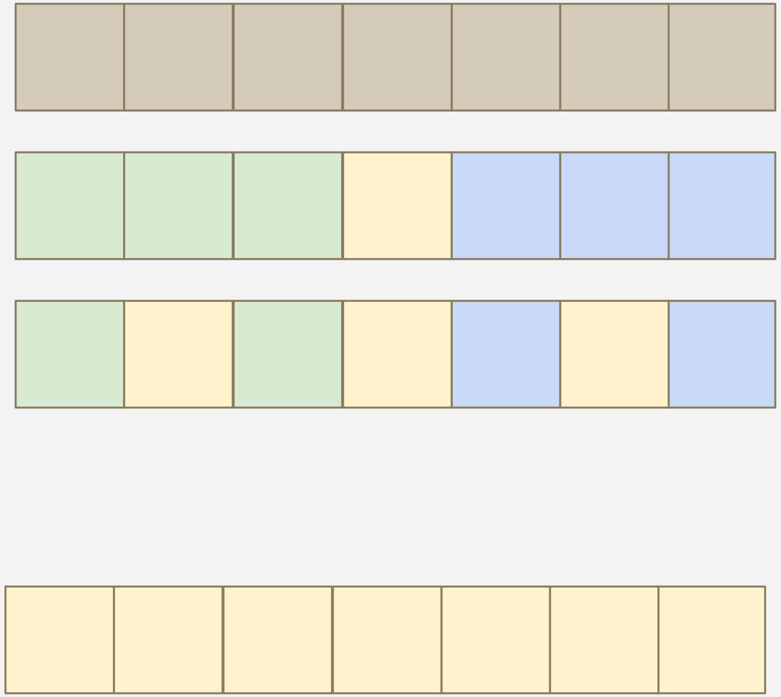


Pseudocode

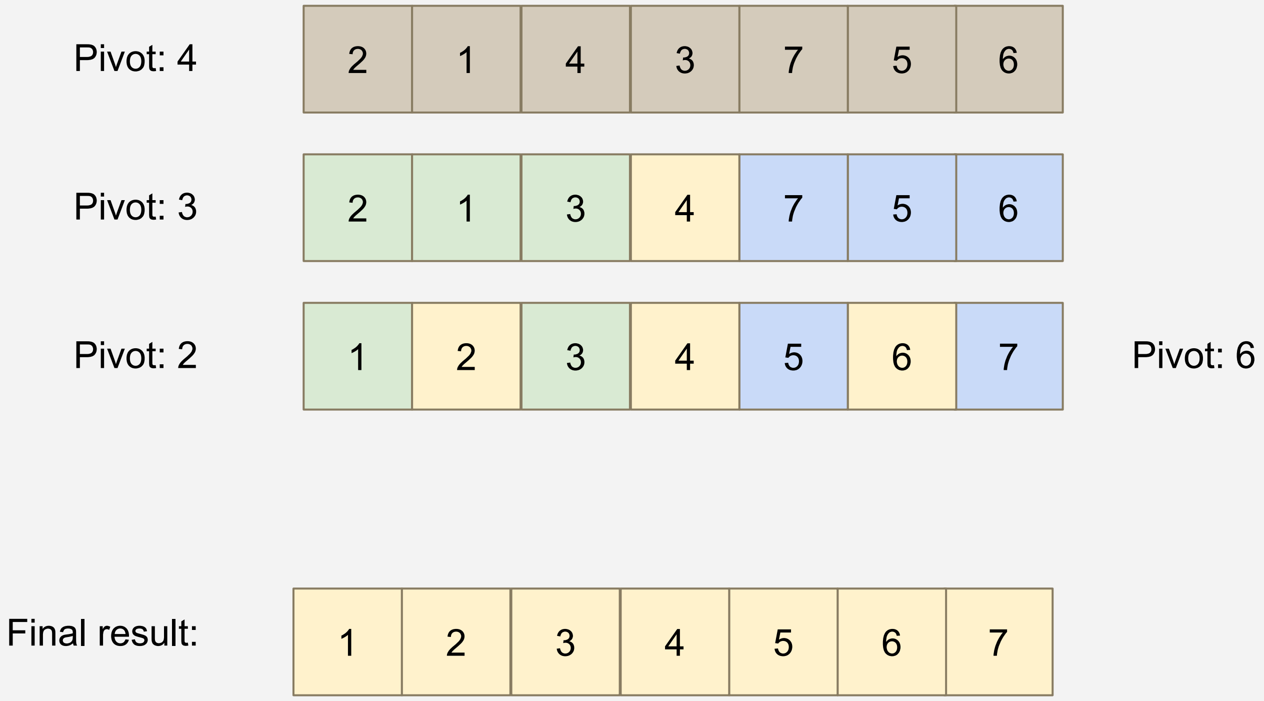
```
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

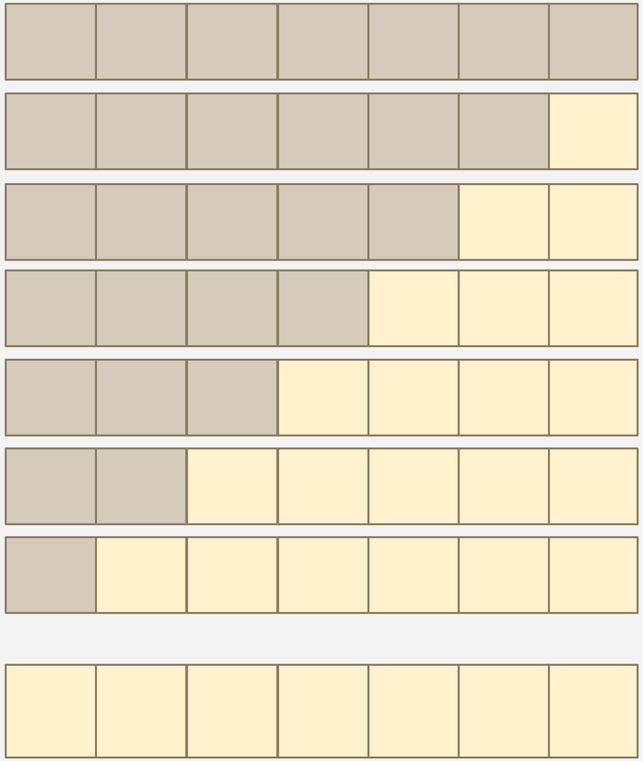
Quicksort Example - Best Case



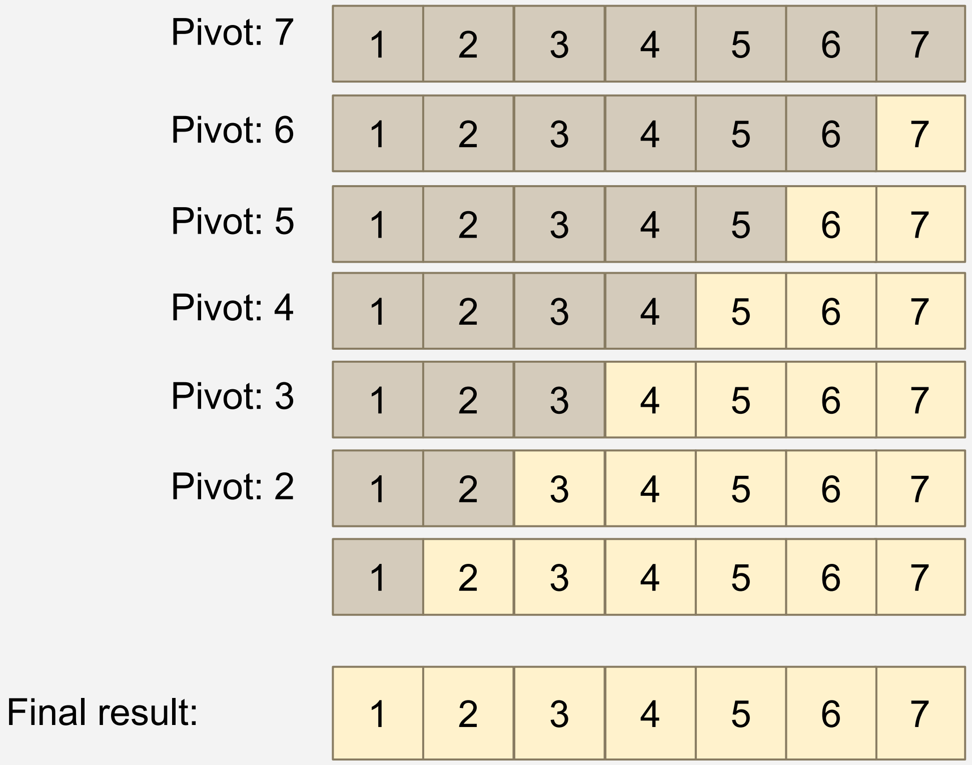
Quicksort Example - Best Case



Quicksort Example - Worst Case



Quicksort Example - Worst Case



Quicksort + Binary Search

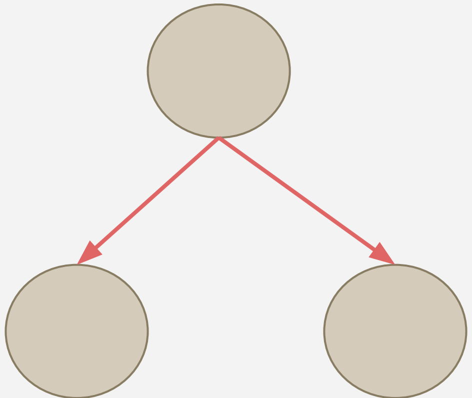
- Complexity
  - Sorting complexity
    - *O(n log n)* (with a bit of luck!)
  - Linear Search complexity
    - *O(log n)*
  - Total
    - *O(n log n) + O(log n)*
- We tried to solve a problem of **efficiently** searching an item in a collection of items
- So far:
  - Data structure: Arrays
  - Sorting: Quicksort
  - Search: Binary Search

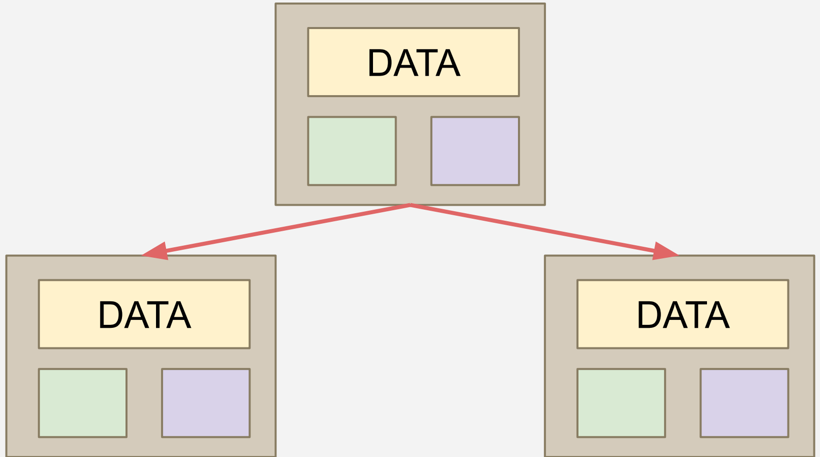
Is this efficient??

- No..
- Why not??
  - Limitations of arrays
    - Data structure with dynamic memory allocation
  - Cannot keep the array sorted efficiently
- What we need?
  - A data structure that allows:
    - Dynamic memory allocation
    - Keep it sorted
    - Efficient search

Trees, Binary Trees, and Binary Search Trees

- Tree
  - a collection of nodes, where each node is a data structure consisting of a value and a list of references to nodes.
- A binary tree
  - is a tree in which each node has at most two children.
  - Left child and right child.





Few more examples

Another example of Logarithmic complexity

```
for(i = 1; i <= n; )
{
     printf("Hello World!!");
     i = i*2;
}
```

- Loop repeats **k** times
- Value of k??
    -

Iterations	Value of i	Power of 2
Iter 1		
Iter 2		
Iter 3		
Iter 4		
Iter k		

Example of Logarithmic complexity

```
for(i = 1; i <= n; )
{
     printf("Hello World!!");
     i = i*2;
}
```

- Loop repeats **k** times
- Value of k??
  - $n = 2^{k-1}$
  - Take log on both sides
    - $\log_2 n = k - 1$
    - $k = \log_2 n + 1$

Iterations	Value of i	Power of 2
Iter 1	i= 1	2^0
Iter 2	i = 2	2^1
Iter 3	i = 4	2^2
Iter 4	i = 8	2^3
Iter k	i = n	2^{k-1}

Evaluating Loops for complexity

```
for(i = 1; i <= n; i++)
{
     printf(“Hello World!!”);
}
```

Total time: O(n)

```
for(i = 1; i <= n; i++)
     for(j = 1; j <= n; j++)
          printf(“Hello World!!”);
```

i = 1	i = 2	i = 3	i = 4	i = n
j runs from 1 to n	j runs from 1 to n	j runs from 1 to n	j runs from 1 to n	j runs from 1 to n

Total time: $O(n^2)$

Evaluating conditionals for complexity

```
if (isValid)
{
  statement1;
  statement2;
} else
{
  statement3;
}
```

Maximum Possible runtime to find out Big O:

Cost of evaluating condition

+

Running time of if part or else part(whichever is the larger)

Evaluating conditionals for complexity

```
if (isValid)
{
  array.sort();
  return true;
} else
{
  return false;
}
```

Maximum Possible runtime to find out Big O:

Cost of evaluating condition (1)

+

Running time of if part or else part(whichever is the larger)

(O(n log n))

Total = O(n log n)

Evaluating function calls for complexity

```
for (i = 0; i < n; i++)
{
  fn1();
  for (j = 0; j < n; j++)
  {
    fn2();
    for (k = 0; k < n; k++)
    {
      fn3();
    }
  }
}
```

Scenarios:
Assume all functions require constant time.

Assume fn1 and fn2 require constant time but fn3 requires $O(n^2)$

Can you make this better?

Sum of numbers

```
int main()
{
  int i, sum = 0, n;
  scanf(“%d”, &n);
  for (i = 0; i < n; i++)
  {
    sum = sum + i;
  }
  printf(“%d”, sum);
  return 0;
}
```

Another example

```
Void fun(int n)
{
      int i, j;
      for(i=1; i<=n/3; i++)
            for(j=1; j<=n; j+=4)
                  printf(“Hello World!\n”);
}
```