COMP26120: Introducing Complexity Analysis (2020/21)

Lucas Cordeiro

lucas.cordeiro@manchester.ac.uk

Asymptotic Performance

- In this course, we care most about ***asymptotic performance***

Asymptotic Performance

- In this course, we care most about ***asymptotic performance***
  - We focus on the **infinite set of large *n*** ignoring small values of *n*
    - The best choice for all, but minimal inputs

Asymptotic Performance

- In this course, we care most about ***asymptotic performance***
  - We focus on the **infinite set of large *n*** ignoring small values of *n*
    - The best choice for all, but minimal inputs
- How does the algorithm behave as the problem size gets vast?
  - Running time
  - Memory/storage requirements
  - Bandwidth/power requirements/logic gates/etc.

Asymptotic Notation

- You should have an **intuitive feel** for asymptotic (big-O) notation:
  - *What does $O(n)$ running time mean? $O(n^2)$? $O(\log_2 n)$?*
  - *How does asymptotic running time relate to asymptotic memory usage?*

Asymptotic Notation

- You should have an **intuitive feel** for asymptotic (big-O) notation:
  - *What does $O(n)$ running time mean? $O(n^2)$? $O(\log_2 n)$?*
  - *How does asymptotic running time relate to asymptotic memory usage?*
- Our first task is to **define this notation more formally**

Search Problem (Arbitrary Sequence)

Input

- *sequence of numbers* $(a_1, \ldots a_n)$
- *search for a specific number* (q)

$a_1, a_2, a_3, \ldots.., a_n; \; q$

Output

- *index or NIL*

j

Search Problem (Arbitrary Sequence)

Input

- *sequence of numbers* $(a_1, \ldots a_n)$
- *search for a specific number* (q)

$a_1, a_2, a_3, \ldots, a_n;\ q \longrightarrow j$

2 5 4 10 7; 5 ⟶ *?*

Output

- *index or NIL*

Search Problem (Arbitrary Sequence)

Input

- *sequence of numbers* $(a_1, \ldots a_n)$
- *search for a specific number (q)*

$a_1, a_2, a_3, \ldots., a_n;\ q \longrightarrow j$

2 5 4 10 7; 5 $\longrightarrow$ *2*

Output

- *index or NIL*

Search Problem (Arbitrary Sequence)

Input

- *sequence of numbers* $(a_1, \ldots a_n)$
- *search for a specific number (q)*

$a_1, a_2, a_3, \ldots, a_n;\ q$ $\longrightarrow$ j

2 5 4 10 7; 5 $\longrightarrow$ *2*

2 5 4 10 7; 9 $\longrightarrow$ *?*

Output

- *index or NIL*

Search Problem (Arbitrary Sequence)

Input

- *sequence of numbers* $(a_1, \ldots a_n)$
- *search for a specific number* (q)

$a_1, a_2, a_3, \ldots, a_n;\ q$

2 5 4 10 7; 5

2 5 4 10 7; 9

Output

- *index or NIL*

j

2

NIL

Linear Search

```
j=1
while j<=length(A) and A[j]!=q
    do j++
if j<=length(A) then return j
else return NIL
```

Linear Search

```
j=1
while j<=length(A) and A[j]!=q
   do j++
if j<=length(A) then return j
else return NIL
```

Linear Search

```
j=1
while j<=length(A) and A[j]!=q
    do j++
if j<=length(A) then return j
else return NIL
```

Linear Search

```
j=1
while j<=length(A) and A[j]!=q
   do j++
if j<=length(A) then return j
else return NIL
```

Linear Search

```
j=1
while j<=length(A) and A[j]!=q
   do j++
if j<=length(A) then return j
else return NIL
```

Linear Search

```
j=1
while j<=length(A) and A[j]!=q
   do j++
if j<=length(A) then return j
else return NIL
```

- Worst case: ***?***, average case: ***?***

Linear Search

```
j=1
while j<=length(A) and A[j]!=q
   do j++
if j<=length(A) then return j
else return NIL
```

- Worst case: ***f(n)=n***, average case: ***n/2***

Linear Search

```
j=1
while j<=length(A) and A[j]!=q
   do j++
if j<=length(A) then return j
else return NIL
```

- Worst case: ***f(n)=n***, average case: ***n/2***
- Can we do better using this approach?

Linear Search

```
j=1
while j<=length(A) and A[j]!=q
   do j++
if j<=length(A) then return j
else return NIL
```

- Worst case: ***f(n)=n***, average case: ***n/2***
- Can we do better using this approach?
  - this is a **lower bound** for the search problem in an **arbitrary sequence**

A Search Problem (Sorted Sequence)

Input

- *sequence of numbers* $(a_1 <= a_2, \ldots, a_{n-1} <= a_n)$
- *search for a specific number (q)*

$a_1, a_2, a_3, \ldots, a_n; \; q$

Output

- *index or NIL*

A Search Problem (Sorted Sequence)

Input

- *sequence of numbers* ($a_1 <= a_2, \ldots, a_{n-1} <= a_n$)
- *search for a specific number* (q)

$a_1, a_2, a_3, \ldots, a_n; \; q$

2 4 5 7 10; 10

Output

- *index or NIL*

j

?

A Search Problem (Sorted Sequence)

Input

- *sequence of numbers ($a_1 <= a_2, \ldots, a_{n-1} <= a_n$)*
- *search for a specific number (q)*

$a_1, a_2, a_3, \ldots, a_n;\ q \longrightarrow j$

$2 \quad 4 \quad 5 \quad 7 \quad 10; \quad 10 \longrightarrow 5$

Output

- *index or NIL*

A Search Problem (Sorted Sequence)

Input

- *sequence of numbers* $(a_1 <= a_2, \ldots, a_{n-1} <= a_n)$
- *search for a specific number (q)*

Output

- *index or NIL*

$a_1, a_2, a_3, \ldots, a_n;\ q \longrightarrow j$

2	*4*	*5*	*7*	*10;*	*10*	*5*
2	*4*	*5*	*7*	*10;*	*8*	*?*

A Search Problem (Sorted Sequence)

Input

- *sequence of numbers* (a_1<=a_2,…, a_{n-1}<=a_n)
- *search for a specific number (q)*

Output

- *index or NIL*

$a_1, a_2, a_3, \ldots, a_n;\ q$ $\longrightarrow$ j

2	*4*	*5*	*7*	*10;*	*10*	*5*
2	*4*	*5*	*7*	*10;*	*8*	*NIL*

A Search Problem (Sorted Sequence)

Input

- *sequence of numbers (a_1<=a_2,…, a_{n-1}<=a_n)*
- *search for a specific number (q)*

Output

- *index or NIL*

$a_1, a_2, a_3, \ldots, a_n; \ q \longrightarrow j$

2	*4*	*5*	*7*	*10;*	*10*	*5*
2	*4*	*5*	*7*	*10;*	*8*	*NIL*

Did the sorted sequence help in the search?

Binary Search

- Assume that the array is **sorted** and then perform **successive divisions**

```
left=1
right=length(A)
do
  j=(left+right)/2
  if A[j]==q then return j
  else if A[j]>q then right=j-1
  else left=j+1
while left<=right
return NIL
```

Binary Search

- Assume that the array is **sorted** and then perform **successive divisions**

```
left=1
right=length(A)
do
  j=(left+right)/2
  if A[j]==q then return j
  else if A[j]>q then right=j-1
  else left=j+1
while left<=right
return NIL
```

Binary Search

- Assume that the array is **sorted** and then perform **successive divisions**

```
left=1
right=length(A)
do
  j=(left+right)/2
  if A[j]==q then return j
  else if A[j]>q then right=j-1
  else left=j+1
while left<=right
return NIL
```

Binary Search

- Assume that the array is **sorted** and then perform **successive divisions**

```
left=1
right=length(A)
do
  j=(left+right)/2
  if A[j]==q then return j
  else if A[j]>q then right=j-1
  else left=j+1
while left<=right
return NIL
```

Binary Search

- Assume that the array is **sorted** and then perform **successive divisions**

```
left=1
right=length(A)
do
   j=(left+right)/2
   if A[j]==q then return j
   else if A[j]>q then right=j-1
   else left=j+1
while left<=right
return NIL
```

Binary Search

- Assume that the array is **sorted** and then perform **successive divisions**

```
left=1
right=length(A)
do
   j=(left+right)/2
   if A[j]==q then return j
   else if A[j]>q then right=j-1
   else left=j+1
while left<=right
return NIL
```

Binary Search

- Assume that the array is **sorted** and then perform **successive divisions**

```
left=1
right=length(A)
do
  j=(left+right)/2
  if A[j]==q then return j
  else if A[j]>q then right=j-1
  else left=j+1
while left<=right
return NIL
```

Binary Search

- Assume that the array is **sorted** and then perform **successive divisions**

```
left=1
right=length(A)
do
  j=(left+right)/2
  if A[j]==q then return j
  else if A[j]>q then right=j-1
  else left=j+1
while left<=right
return NIL
```

Binary Search

- Assume that the array is **sorted** and then perform **successive divisions**

```
left=1
right=length(A)
do
  j=(left+right)/2
  if A[j]==q then return j
  else if A[j]>q then right=j-1
  else left=j+1
while left<=right
return NIL
```

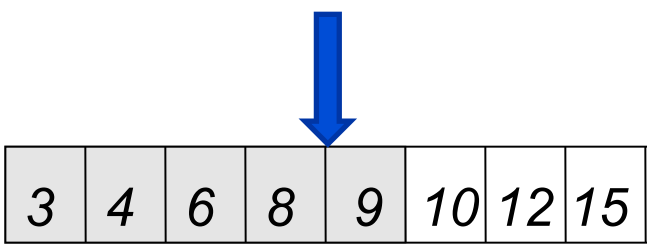
Binary Search Analysis

- How many times is the loop executed?

3	*4*	*6*	*8*	*9*	*10*	*12*	*15*

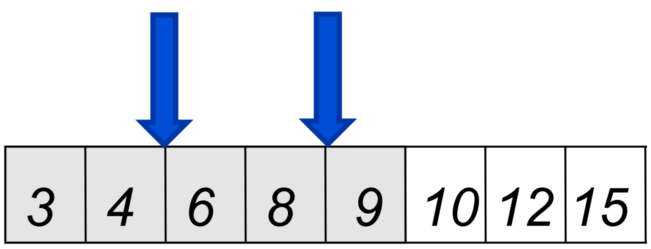
Binary Search Analysis

- How many times is the loop executed?



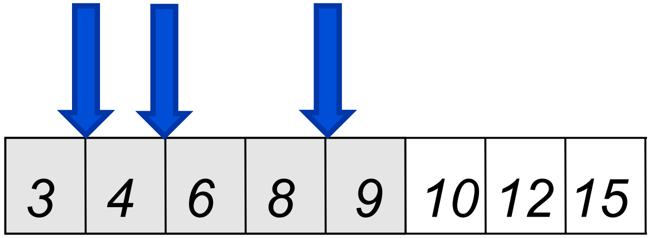
Binary Search Analysis

- How many times is the loop executed?



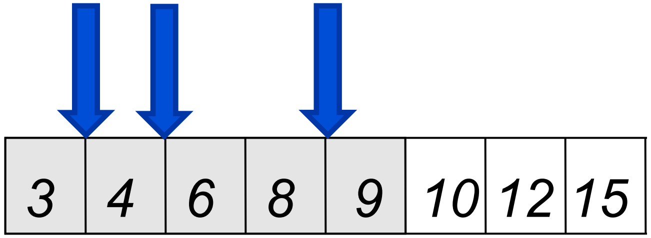
Binary Search Analysis

- How many times is the loop executed?



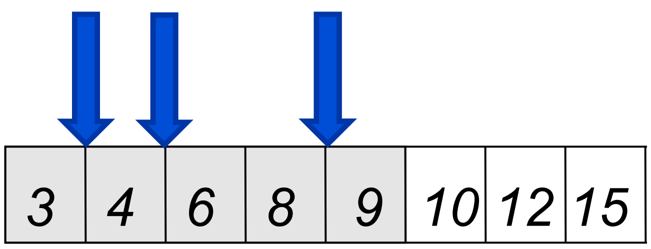
Binary Search Analysis

- How many times is the loop executed?
  - At each interaction, the number of positions **n** is cut in half
  - How many times do we cut in half **n** to reach 1?



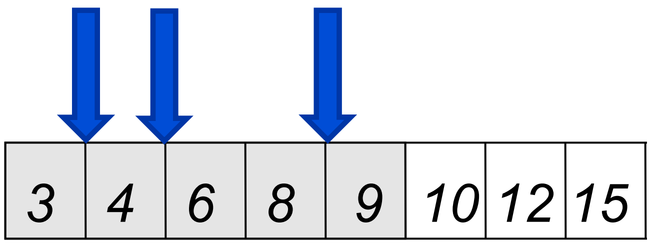
Binary Search Analysis

- How many times is the loop executed?
  - At each interaction, the number of positions **n** is cut in half
  - How many times do we cut in half **n** to reach 1?
    - $\mathbf{lg_2\ n}$



Binary Search Analysis

- How many times is the loop executed?
  - At each interaction, the number of positions **n** is cut in half
  - How many times do we cut in half **n** to reach 1?
    - o **lg_2 n**



$$\lg_2 n = x \Leftrightarrow n = 2^x$$

$$\lg_2 8 = 3$$

Analysis of Algorithms (COMP15111)

- We perform the analysis concerning a **computational model**

Analysis of Algorithms (COMP15111)

- We perform the analysis concerning a **computational model**
- We will usually use a generic uniprocessor **random-access machine** (RAM)

Analysis of Algorithms (COMP15111)

- We perform the analysis concerning a **computational model**
- We will usually use a generic uniprocessor **random-access machine** (RAM)
  - All memory **equally expensive** to access

Analysis of Algorithms (COMP15111)

- We perform the analysis concerning a **computational model**
- We will usually use a generic uniprocessor **random-access machine** (RAM)
  - All memory **equally expensive** to access
  - Instructions executed one after another (**no concurrent operations**)

Analysis of Algorithms (COMP15111)

- We perform the analysis concerning a **computational model**
- We will usually use a generic uniprocessor **random-access machine** (RAM)
  - All memory **equally expensive** to access
  - Instructions executed one after another (**no concurrent operations**)
  - All reasonable instructions take **unit time**
    - Except, of course, function calls

Analysis of Algorithms (COMP15111)

- We perform the analysis concerning a **computational model**
- We will usually use a generic uniprocessor **random-access machine** (RAM)
  - All memory **equally expensive** to access
  - Instructions executed one after another (**no concurrent operations**)
  - All reasonable instructions take **unit time**
    - Except, of course, function calls
  - Constant word size
    - Unless we are explicitly manipulating bits

Input Size

- **Time** and **space** complexity
  - This is generally a **function of the input size**
    - E.g., sorting, multiplication

Input Size

- **Time** and **space** complexity
  - This is generally a **function of the input size**
    - E.g., sorting, multiplication
  - How we characterize input size depends:
    - **Sorting:** number of input items
    - **Multiplication:** total number of bits
    - **Graph algorithms:** number of nodes and edges
    - Etc.

Running Time

- Number of **primitive steps** that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
    - o f = (g + h) – (i + j)
    - o y = m * x + b
    - o c = 5 / 9 * (t - 32)
    - o z = f(x) + g(y)

Running Time

- Number of **primitive steps** that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
    - o f = (g + h) – (i + j)
    - o y = m * x + b
    - o c = 5 / 9 * (t - 32)
    - o z = f(x) + g(y)
- We can be more exact if needed

```
add t0, g, h # temp t0 = g + h
add t1, i, j # temp t1 = i + j
sub f, t0, t1 # f = t0 - t1
```

Analysis

- **Worst case**
  - Provides an **upper bound** on running time
  - An (absolute) guarantee

Analysis

- **Worst case**
  - Provides an **upper bound** on running time
  - An (absolute) guarantee
- **Average case**
  - Provides the expected running time
  - Very useful, but treat with care: what is “average”?
    - Random (equally likely) inputs
    - Real-life inputs