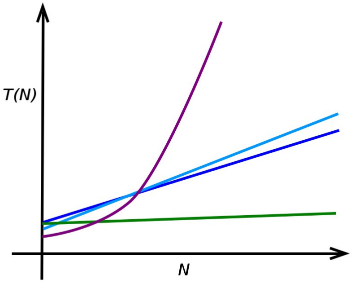
# CO580 Algorithms

Dr Timothy Kimber

January 2016



# Course Outline

This course will cover

- The role of algorithms in computing
- How to design algorithms
- How to evaluate algorithms
- (Lots of) algorithms used in specific settings

This course will be

- Practical – it should take your programming to the next level
- Somewhat mathematical

The main language used will be Java, although a good algorithm can usually be implemented in many languages (even Prolog!)

# Course Outline

## The lecturer

- I am a Teaching Fellow in DoC
- I have a PhD in Computational Logic
- I teach Prolog to the MAC and Specialism classes and the Intro to Java programming

## The structure

- 27 hours of lectures and lab-based tutorials (weeks 2–10)
- One or two assessed exercises (10%)
- A 2-hour written examination *next term* (90%)

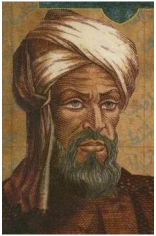
## Books

- *Introduction to Algorithms*, Cormen et al., 3rd edn, 2009.
- *Algorithms*, Sedgewick & Wayne 4th edn, 2011.

# What Is An Algorithm?

**Algorithm** a procedure for solving **a mathematical problem** in a finite number of steps that frequently involves repetition of an operation; *broadly* : a step-by-step procedure for solving a problem or accomplishing some end (especially by a computer)

*http://www.merriam-webster.com/dictionary/algorithm*



Muḥammad ibn Mūsā al-Khwārizmī (780–850)

# Some Mathematical Problems

**Example** *(AgeCalc)*

**Given:** a person's current age $Age$; the current year $Year$; another year $AnotherYear$

**Find:** the person's age in $AnotherYear$

A Solution: $Age + (AnotherYear - Year)$

**Example** *(Greatest Common Divisor)*

**Given:** two integers $X$ and $Y$

**Find:** an integer $Z$ such that: $Z$ is a divisor of both $X$ and $Y$; there is no integer $Z' > Z$ that is also a divisor of both $X$ and $Y$

A Solution: Euclid's Algorithm (300 BC)

# Algorithms, Programs, Correctness

- So, an algorithm is the underlying solution that is implemented by a program.
- We will see how algorithms can be analysed independently from any particular implementation and what makes a "good" algorithm.

## Properties of Algorithms

- You have been focusing on writing programs that satisfy one key requirement: that they implement a correct algorithm.

**Definition (Correct Algorithm)**

A procedure is a *correct algorithm* for a problem iff, for every input instance of the problem, it halts with the correct output [Cormen p6]

# Beyond Correctness

- This course goes beyond correctness to look at another vital property of algorithms: the resources they use
- Resources:

| | |
|---|---|
| space | main memory used |
| time | number of CPU cycles used |

- Commonly we are most interested in time (speed)
- Space and time are often traded off against one another. Using memory to store extra information can save a lot of time

# Time

- So, how long will my algorithm take to 'run'?
- Consider this example:

**Example** *(List Search)*

Input: the sequence (list) $L = \langle a_1, ..., a_N \rangle$ of integers, and an integer $k$

Output: *True* if $k$ is in $L$, *False* otherwise

Simplification: assume $L$ is ordered

# Simple List Search

**Simple Search (Input: list $L$ and value $k$)**

- For each element $e$ in $L$
  - If $e == k$ Output *True* and HALT
- Output *False* and HALT

Questions to answer:

- Is Simple Search correct?
- How long will it take to run?

# Simple Search

$k = 21$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $L$ | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |

# Simple Search

$k = 21$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $L$ | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |

# Simple Search

$k = 21$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $L$ | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |

# Simple Search

$k = 21$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $L$ | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |

# Simple Search

$k = 21$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $L$ | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |

# Simple Search

$k = 21$

| $L$ | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |
|---|---|---|---|---|---|---|---|---|

- Output: True
- A very simple example, but how we are going to analyse it is the important part

# Java Implementation

```
private static boolean search(int[] a, int k) {

    int N = a.length;

    for (int i = 0; i < N; i++) {

        if (a[i] == k)  { return true; }

    }

    return false;
}
```

- So, how long will this Java implementation take to run?

# Java Implementation

```
private static boolean search(int[] a, int k) {

    int N = a.length;

    for (int i = 0; i < N; i++) {

        if (a[i] == k)  { return true; }

    }

    return false;
}
```

- So, how long will this Java implementation take to run?
- Running time will depend on: compiler, processor, ..., size of input, and type of input
- We need to decide what input case to consider
- What would be best, average or worst case inputs?

# Simple Search: Worst Case Analysis

- For worst case input Simple Search is a 'linear' algorithm
- We construct a formula to represent the running time for an input of 'size' $N$: $T(N) = \sum_{i=1}^{n} c_i t_i$
- The cost (time taken) for line $i$ is represented by $c_i$. For a given processor etc. each $c_i$ is constant and small.

```
private static boolean search(int[] a, int k) {     // COST     TIMES
    int N = a.length;                               //  c1        1
    for (int i = 0; i < N; i++) {                   //  c2        N + 1
        if (a[i] == k)                              //  c3        N
          { return true; }                          //  c4        0
    }
    return false;                                   //  c5       1
}
```

# Runtime Analysis

- So, the running time for a worst case input of size $N$ is

$$T(N) = c_1 + (N+1)c_2 + Nc_3 + c_5$$

- or

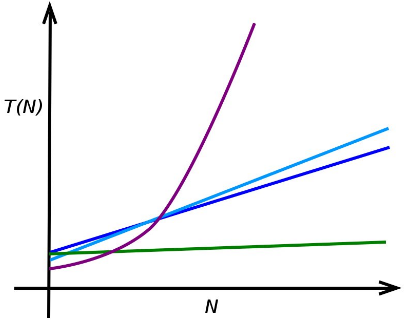
$$T(N) = (c_1 + c_2 + c_5) + (c_2 + c_3)N$$

- or

$$T(N) = aN + b$$

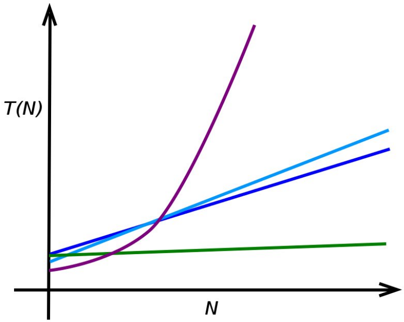
- We can empirically determine $a$ and $b$ for a specific program, machine. This allows performance to be predicted, improved etc.
- However, the general formula applies for any implementation
- The important (but unsurprising) result is that $T(N)$ is a linear function of $N$. The running time is directly proportional to $N$.

# The Algorithm Battlefield




- The running time of an algorithm is some function of $N$
- For small $N$ all algorithms are
  - fast
  - roughly the same

# The Algorithmm Battlefield



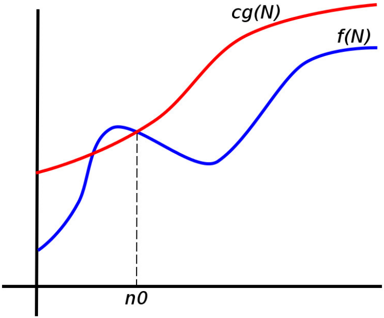

For large $N$:

- Algorithm performance resolves into classes according to the highest order term in $N$ (order $N$, order $N^2$ etc.)
- $T(N) = aN^2 + bN \gg T'(N) = cN$ regardless of $a, b, c$ (Why?)
- This provides clear goals for algorithm design

# Asymptotic Notation

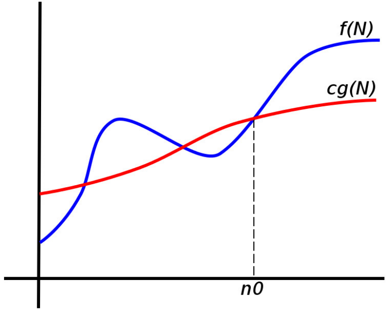
- "For large $N$" means as $N$ grows without bound or asymptotically.
- The asymptotic growth of a function (such as $T(N)$ for some algorithm) is classified using $\Theta$, $O$ and $\Omega$ notations.
- These notations define bounds on the growth of a function $f(N)$ with respect to some other function $g(N)$.
- $f(N)$ is
  - $O(g(N))$ if $g(N)$ is an asymptotic upper bound for $f(N)$
  - $\Omega(g(N))$ if $g(N)$ is an asymptotic lower bound for $f(N)$
  - $\Theta(g(N))$ if $g(N)$ is an asymptotically tight bound for $f(N)$
- The asymptotic bound of its running time (most often $O(g(N))$) is the most common way to characterise an algorithm

# Big O: Upper Bound



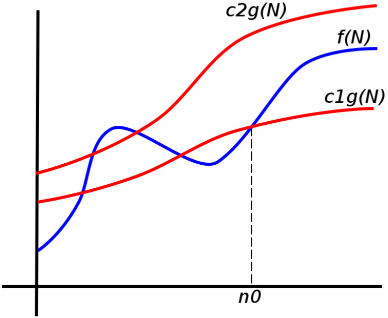

$$O(g(N)) = \left\{ f(N) \mid \begin{array}{l} \text{there are positive constants } c \text{ and } n_0 \\ \text{such that } 0 \leq f(N) \leq c\, g(N) \text{ for all } N \geq n_0 \end{array} \right\}$$

# Big Omega: Lower Bound




$$\Omega(g(N)) = \left\{ f(N) \;\middle|\; \begin{array}{l} \text{there are positive constants } c \text{ and } n_0 \\ \text{such that } 0 \leq c\,g(N) \leq f(N) \text{ for all } N \geq n_0 \end{array} \right\}$$

# Big Theta: Tight Bound




$$\Theta(g(N)) = \left\{ f(N) \mid \begin{array}{l} \text{there are positive constants } c_1,\ c_2 \text{ and } n_0 \\ \text{such that} \\ 0 \leq c_1\, g(N) \leq f(N) \leq c_2\, g(N) \text{ for all } N \geq n_0 \end{array} \right\}$$

# Asymptotic Runtime Analysis

## For Simple Search

- The worst case performance is given by $T(N) = aN + b$
- This function is $\Omega(N)$, $O(N)$ and $\Theta(N)$, written (abusively)
  - $T(N) = \Omega(N)$
  - $T(N) = O(N)$
  - $T(N) = \Theta(N)$
- Therefore, the running time for any input is also $O(N)$
- Is the running time for any input also $\Omega(N) and\ \Theta(N)$?

## In General

- $f(N) = \Theta(N) \iff f(N) = O(N)$ and $f(N) = \Omega(N)$
- $f(N) = O(N^x) \implies f(N) = O(N^y)$ for all $y > x$

# Highest Order Terms

If running time is independent of $N$, we say the algorithm runs in constant time and write

- $T(N) = \Theta(1)$

As we would expect, in a polynomial the term with the largest exponent dominates.

**Definition (Polynomial)**

A polynomial of degree $d$ (for $d \geq 1$) is a function $p(N)$ of the form

$$p(N) = a_0 + a_1 N + a_2 N^2 + \cdots + a_d N^d$$

in which $a_d \neq 0$. The polynomial is asymptotically positive iff $a_d > 0$.

If $T(N)$ is an asymptotically positive polynomial of degree $d$, then

- $T(N) = \Theta(N^d)$

# Highest Order Terms

Exponential functions include a term of the form $a^N$.

- If $a > 1$ then the function grows faster than any polynomial

Polylogarithmic functions include a term of the form $(\log_2 N)^k$.

- Recall: $b^{\log_b a} = a$
- $k$ is a constant
- Changing base just multiplies by a constant
- Any positive polynomial grows faster than any polylogarithm

# Binary List Search

- So, we have a $O(N)$ search algorithm. Can we do any better?
- Do we have to look at every element? Since list is ordered we can eliminate whole chunks at a time.

**Binary Search (Input: list $L$, value $k$)**

- Repeat
    - Choose an element $e$ near the middle of the list
    - If $e == k$, Output *True* and HALT
    - Otherwise, if $k > e$, continue searching elements after $e$
    - Otherwise continue searching elements before $e$
- Until there is nothing left to search
- Output *False* and HALT

- Is Binary Search correct?

# Binary List Search

$k = 1$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $L$ | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |

# Binary List Search

$k = 1$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $L$ | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |

# Binary List Search

$k = 1$

| $L$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |

# Binary List Search

$k = 1$

# Binary List Search

$k = 1$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $L$ | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |

# Binary List Search

$k = 1$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $L$ | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |

# Binary List Search

$k = 1$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $L$ | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |

# Binary List Search

$k = 1$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $L$ | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |

- Output: False

# Java Implementation

```java
private static boolean binarySearch(int k, int[] a) {

    int l = 0, m, n = a.length;       // array indexes

    while (n > l) {
        m = l + (n - l) / 2;

        if      (k == a[m]) { return true; }
        else if (k > a[m])  { l = m + 1;   }
        else                { n = m;       }
    }

    return false;
}
```

- What is the worst case performance?

# Binary List Search: Java

$k = 1$

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | |
|---|---|---|---|---|---|---|---|---|
| a | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |

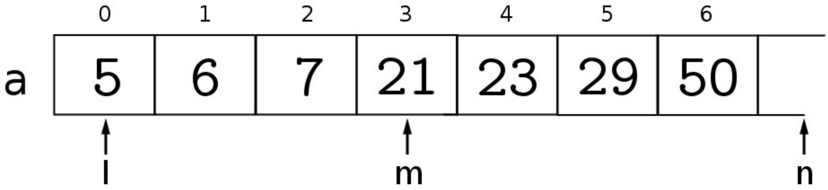
# Binary List Search: Java

$k = 1$

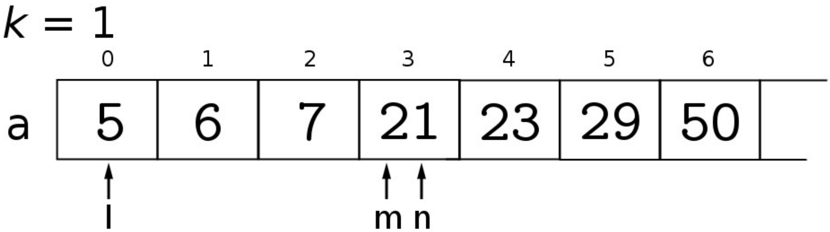
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | |
|---|---|---|---|---|---|---|---|---|
| a | 5 | 6 | 7 | 21 | 23 | 29 | 50 | |
| | ↑ l | | | | | | | ↑ n |

# Binary List Search: Java

$k = 1$

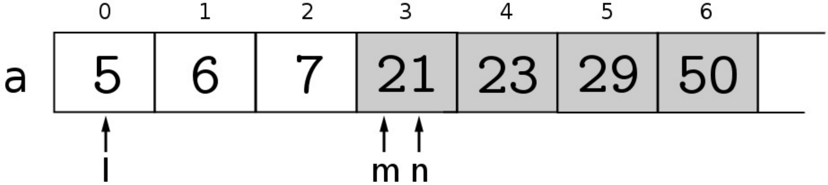


# Binary List Search: Java

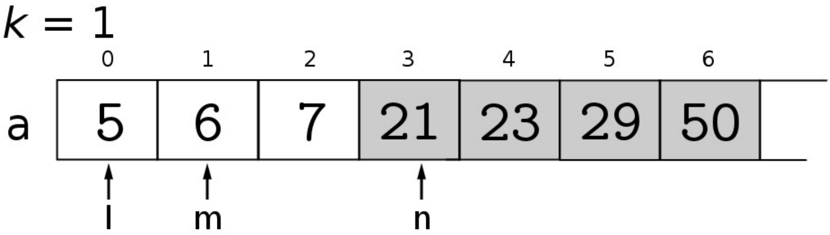


# Binary List Search: Java

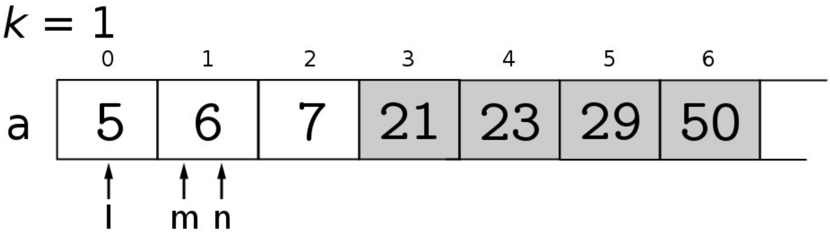
$k = 1$



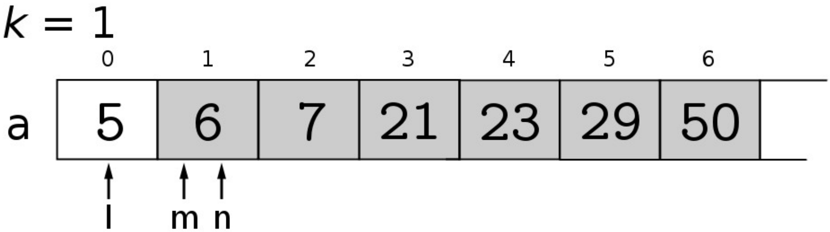
# Binary List Search: Java



# Binary List Search: Java

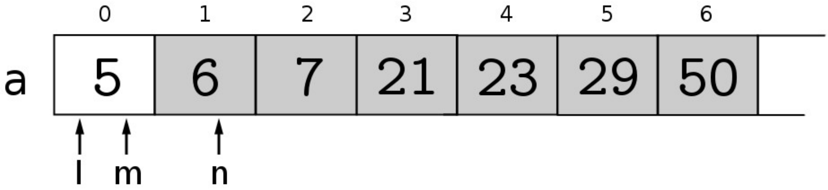


# Binary List Search: Java

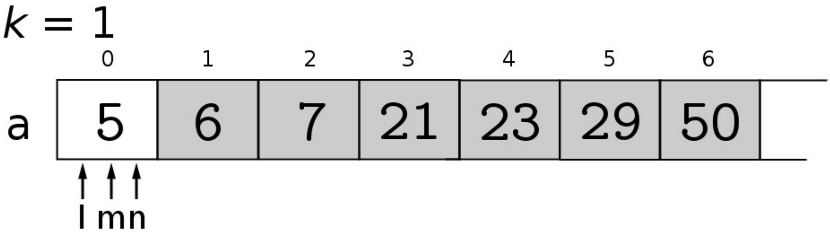


# Binary List Search: Java

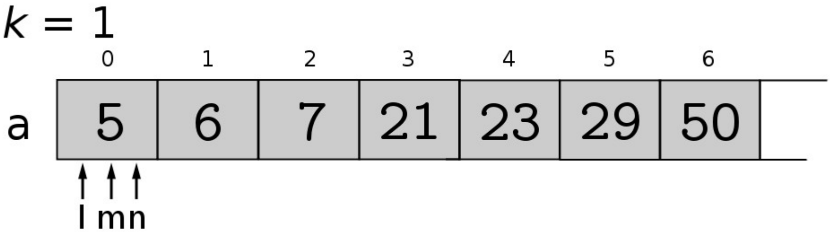
$k = 1$



# Binary List Search: Java

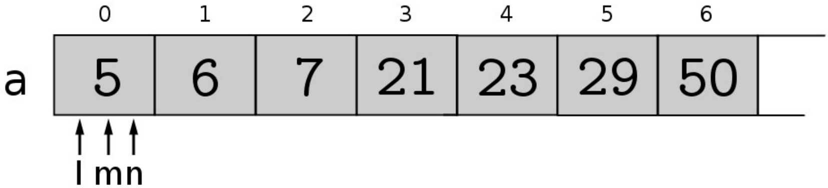


# Binary List Search: Java



# Binary List Search: Java

$k = 1$




- `false`

# Java Implementation

```
private static boolean binarySearch(int k, int[] a) {

    int l = 0, m, n = a.length;       // array indexes

    while (n > l) {
        m = l + (n - l) / 2;

        if      (k == a[m]) { return true; }
        else if (k > a[m])  { l = m + 1;   }
        else                { n = m;       }
    }

    return false;
}
```

- What is the worst case performance?

# Binary Search: Worst Case Analysis

```
private static boolean binarySearch(int k, int[] a) {  // COST    TIMES

    int l = 0, m, n = a.length;                        //  c1      1

    while (n > l) {                                    //  c2      t(N) + 1
        m = l + (n - l) / 2;                           //  c3      t(N)

        if      (k == a[m])                            // c4       t(N)
                              { return true; }         // c5       0
        else if (k > a[m])                             // c6       t(N)
                              { l = m + 1;   }         // c7       0
        else
                              { n = m;       }         // c8       t(N)
    }

    return false;                                      // c9       1
}
```

- For now just let number of loop iterations required to search a list of size $N$ be $t(N)$

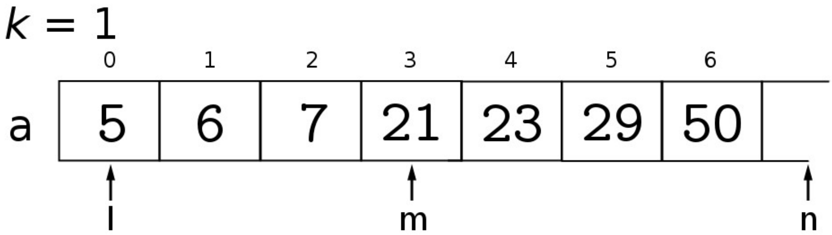
# Recurrence Equations

- $t(N)$ can be expressed in the form of a recurrence equation:

$$t(N) = \begin{cases} 1 & \text{, if } N = 1 \\ 1 + t(N') & \text{, if } N > 1 \end{cases}$$

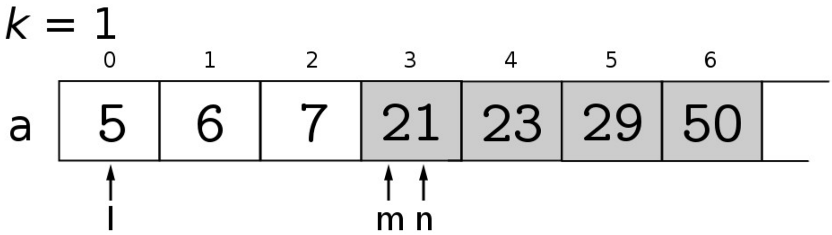
- $N'$ represents the number of elements left to search if we start with $N$ and execute the loop once
- Not quite a recurrence yet - we need to replace $N'$ with an expression involving $N$
- What is $N'$ (in the worst case)?

# Worst Case for N'



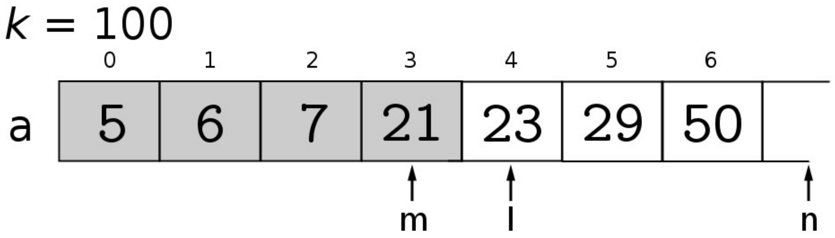

- Observe: `m` is always placed at `l` $+ \lfloor N/2 \rfloor$
- if `k < a[m]` we have $\lfloor N/2 \rfloor$ elements left
- if `k > a[m]` we have
  - $\lfloor N/2 \rfloor$ elements left if $N$ is odd
  - $\lfloor N/2 \rfloor - 1$ elements left if $N$ is even
- So the most elements we have left to search is $N' = \lfloor N/2 \rfloor$

# Worst Case for N'



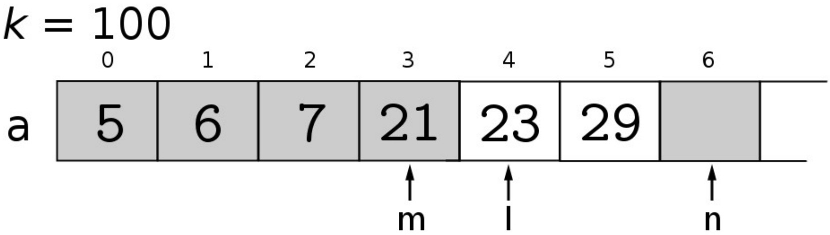

- Observe: `m` is always placed at `l` $+ \lfloor N/2 \rfloor$
- if `k < a[m]` we have $\lfloor N/2 \rfloor$ elements left
- if `k > a[m]` we have
  - $\lfloor N/2 \rfloor$ elements left if $N$ is odd
  - $\lfloor N/2 \rfloor - 1$ elements left if $N$ is even
- So the most elements we have left to search is $N' = \lfloor N/2 \rfloor$

# Worst Case for N'




- Observe: `m` is always placed at `l` $+ \lfloor N/2 \rfloor$
- if `k < a[m]` we have $\lfloor N/2 \rfloor$ elements left
- if `k > a[m]` we have
  - $\lfloor N/2 \rfloor$ elements left if $N$ is odd
  - $\lfloor N/2 \rfloor - 1$ elements left if $N$ is even
- So the most elements we have left to search is $N' = \lfloor N/2 \rfloor$

# Worst Case for N'




- Observe: `m` is always placed at `l` $+ \lfloor N/2 \rfloor$
- if `k < a[m]` we have $\lfloor N/2 \rfloor$ elements left
- if `k > a[m]` we have
  - $\lfloor N/2 \rfloor$ elements left if $N$ is odd
  - $\lfloor N/2 \rfloor - 1$ elements left if $N$ is even
- So the most elements we have left to search is $N' = \lfloor N/2 \rfloor$

# Solving the Recurrence

- So, the actual recurrence is

$$t(N) = \begin{cases} 1 & \text{, if } N = 1 \\ 1 + t(\lfloor N/2 \rfloor) & \text{, if } N > 1 \end{cases}$$

- Now we need to solve the recurrence, so that we can determine the actual value of $t(N)$ when $N > 1$
- Simplification: assume $N$ is some power of 2
- Then $\lfloor N/2 \rfloor = N/2$

# Solving the Recurrence

- One technique is to telescope the recurrence
- Substituting terms on the right hand side we have

$$t(N) = 1 + t(N/2) \quad (1)$$
$$t(N) = 1 + 1 + t(N/4) \quad (2)$$
$$t(N) = 1 + 1 + 1 + t(N/8) \quad (3)$$

- We need to know how many 1s we have when the expression reaches

$$t(N) = 1 + \cdots + t(1)$$

- From (1–3) above we can see the answer is $j$ where $2^j = N$
- So, $j = log_2(N)$ and $t(N) = 1 + log_2(N)$, for all $N$
- If we drop the assumption $t(N)$ will still contain a $\log_2 N$ term

# Binary Search: Worst Case Analysis

```
private static boolean binarySearch(int k, int[] a) {  // COST    TIMES

    int l = 0, m, n = a.length;                         //  c1      1

    while (n > l) {                                     //  c2      t(N) + 1
        m = l + (n - l) / 2;                            //  c3      t(N)

        if      (k == a[m])                             // c4       t(N)
                            { return true; }            // c5       0
        else if (k > a[m])                              // c6       t(N)
                            { l = m + 1;   }            // c7       0
        else
                            { n = m;       }            // c8       t(N)
    }

    return false;                                       // c9       1
}
```

$$T(N) = (c_1 + c_2 + c_9) + (c_2 + c_3 + c_4 + c_6 + c_8)(1 + \log_2(N))$$
$$T(N) = a \log_2(N) + b$$

- So, $T(N) = \Theta(\log_2(N))$

# Comparing the Algorithms

- Suppose we have implementations of the two algorithms:
  - Simple Search (excellent programmer): uses $5N$ instructions
  - Binary Search (average programmer): uses $100\log_2(N)$ instructions
- Simple Search uses 5000 instructions to search 1000 numbers.
- Binary Search can search $2^{5000/100} = 1.1 \times 10^{15}$ numbers in the same time! (if we had enough memory)
- In 1971 the first microprocessor ran at 740 kHz. If that machine ran Binary Search it could search a list of 10 million numbers faster than our lab machines (3.4 GHz) using Simple Search.
- Of course, both computers would still take much less than a second.
- However, 'naive' algorithms for some problems are often $O(N^2)$ or worse. In this case running times can easily be hours or days, even on today's hardware.

# Divide and Conquer

- The Binary Search algorithm uses a divide and conquer approach.
- Divide and conquer reduces the main problem into a number of smaller subproblems (one in this case).
- The solutions to the subproblems may or may not need to be combined to produce the final solution.
- Divide and conquer often produces logarithmic performance.
- Divide and conquer is a technique that can be widely employed when designing algorithms. We shall see other such general algorithmic schemes.