# Chapter 11:
# Brutes Forces, Secretaries, and Dichotomies

Panos Louridas

Athens University of Economics and Business
Real World Algorithms
A Beginners Guide
The MIT Press

# Outline


1. Sequential Search
2. The Matthew Effect and Power Laws
3. Self-Organizing Search
4. The Secretary Problem
5. Binary Search
6. Representing Integers in Computers
7. Binary Search Revisited
8. Comparison Trees

# Searching in Unordered Elements

- If we are looking for a card in a shuffled deck of cards, there is not much we can do apart from checking each one of them in turn.
- We can follow any strategy:
  - From the beginning to the end.
  - From the end towards the beginning.
  - Taking a different card each time, at random.
- If the deck is unordered, any strategy that checks each one once is as good as any other.
- When we do an exhaustive search through our material we are following a *brute force* approach, because we do not use any kind of cleverness.

# Sequential Search

- Sequential search is the most obvious strategy.
- We check the first element.
- If it is the one we are looking for, we are done.
- Otherwise we take the next one, and so on, until we find the one we are looking for, or we run out of elements.

# Sequential Search Algorithm

**Algorithm:** Sequential search.

`SequentialSearch`$(A, s) \rightarrow i$

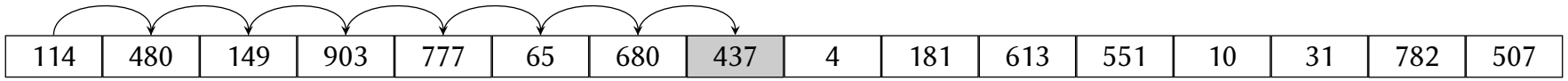
**Input:** $A$, an array of items

$s$, an element we are searching for

**Output:** $i$, the position of $s$ in $A$ if $A$ contains $s$, or $-1$ otherwise

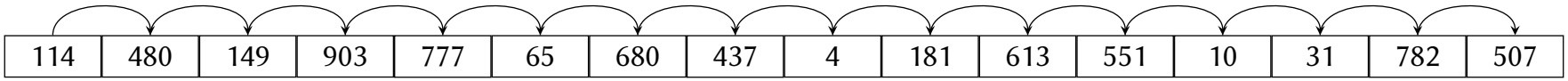
```
1  for i ← 0 to |A| do
2      if Matches(A[i], s) then
3          return i
4  return −1
```

# Example of Successful Search




Searching for 437.

# Unsuccessful Search




Searching for 583.

# Records

- We store data in computers in the form of *records*.
- A record is a set of *attributes* or *fields* together with their associated *values*.

# Examples of Records

| first name: John |
| --- |
| surname: Doe |
| passport no: AI892495 |
| age: 36 |
| occupation: teacher |

| real: 3.14 |
| --- |
| imaginary: 1.62 |

# Keys

- We call *key* one or more attributes that identify a record.
- If the key consists of more than one attributes it is called a *compound key* or *composite key*.
- A record may have more than one keys. In that case, we designate the one that we choose to work with primarily as the *primary key*. The other keys are *secondary keys*.
- For example, a student identification number assigned by the institution is a key for a student's record. Another key could be the combination of name, surname, birth day, and department. We could appoint the student identification number as the primary key.

# The Matches Function

- We use the Matches function to check if an element is the one we are looking for.
- Matches$(x, y)$ returns TRUE if the key of $x$ is equal with the key of $y$, otherwise it returns FALSE.

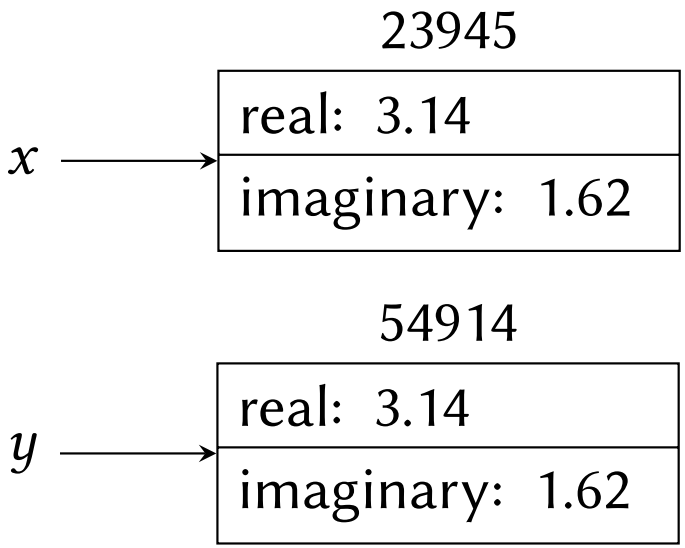
# Equality in Computers

There are two notions of equality in computers. Suppose we have two variables $x$ and $y$.

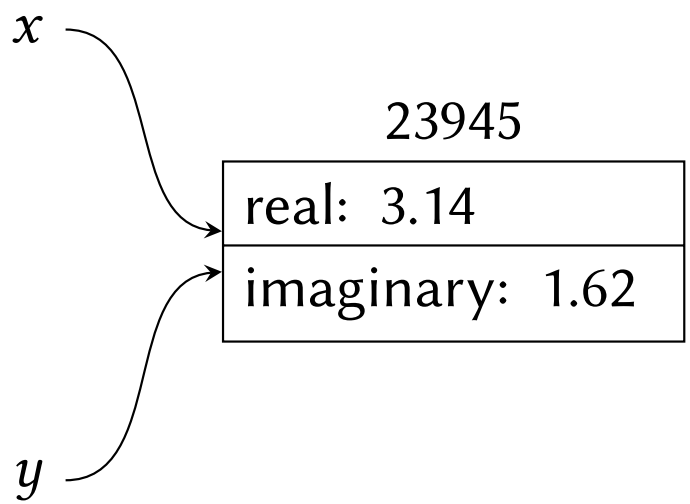
- $x$ and $y$ may point to two records with the same keys (and contents).
- $x$ and $y$ may point to the same record.

The second case is called *strict equality* and the comparison is called *strict comparison*. It is faster, but it is not usually the one we want.

# Two Kinds of Equality




Same contents, but $x \neq y$.




Same record, $x = y$.

# Sequential Search Algorithm

**Algorithm:** Sequential search.

`SequentialSearch`$(A, s) \rightarrow i$

**Input:** $A$, an array of items

$s$, an element we are searching for

**Output:** $i$, the position of $s$ in $A$ if $A$ contains $s$, or $-1$ otherwise

```
1  for i ← 0 to |A| do
2      if Matches(A[i], s) then
3          return i
4  return −1
```

# Performance of Sequential Search

- Suppose that $|A| = n$ and that the elements of $A$ are in random order.
- Then $s$ may be in any position of $A$ with the same probability.
- It may be at the first position with probability $1/n$, at the second position with probability $1/n$, up to the last position with probability $1/n$.
- If it is the first item, the loop will be executed once. If it is the second item, it will be executed twice. If we do not find it at all, the loop will be executed $n$ times.

# Performance of Sequential Search (Contd.)

- On average the loop will be executed:

$$\frac{1}{n} \times 1 + \frac{1}{n} \times 2 + \cdots + \frac{1}{n} \times n = \frac{1 + 2 + \cdots + n}{n} = \frac{n+1}{2}$$

  because:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

- So the performance of a successful search is $O((n+1)/2) = O(n)$.
- The performance of an unsuccessful search is $\Theta(n)$.

# Can We Improve Sequential Search?

- Suppose that the elements of $A$ are not in random order, but in the order of their popularity.
- The first element is the one we are looking for most of the time, the second one is the second most popular one, and so on.
- Then sequential search will be improved.
- But does this happen?

# Outline


1. Sequential Search
2. The Matthew Effect and Power Laws
3. Self-Organizing Search
4. The Secretary Problem
5. Binary Search
6. Representing Integers in Computers
7. Binary Search Revisited
8. Comparison Trees

# The Matthew Effect

## Matthew, 25:29

*For unto every one that hath shall be given, and he shall have abundance: but from him that hath not shall be taken even that which he hath.*

# Examples of the Matthew Effect

- Wealth distribution.
- City sizes.
- Number of hyperlinks.
- Bestsellers.
- ...

# Zipf's Law

In linguistics, the phenomenon was observed by the Harvard linguist George Kingsley Zipf (1935, 1949), who observed that the $i$th most common word in a language with a vocabulary of $n$ words appears with probability approximately $1/i$. Zipf's law states that:

$$P(i) = \frac{1}{i}\frac{1}{H_n}$$

where:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

# Harmonic Numbers

- $H_n$ is the $n$th *harmonic number*.
- The name comes from overtones, or harmonics, in music.
- A string vibrates in a fundamental wavelength and also harmonics that are $1/2$, $1/3$, $1/4$,... of the fundamental wavelength.
- The sum $H_n$ when $n = \infty$ is called *harmonic series*:

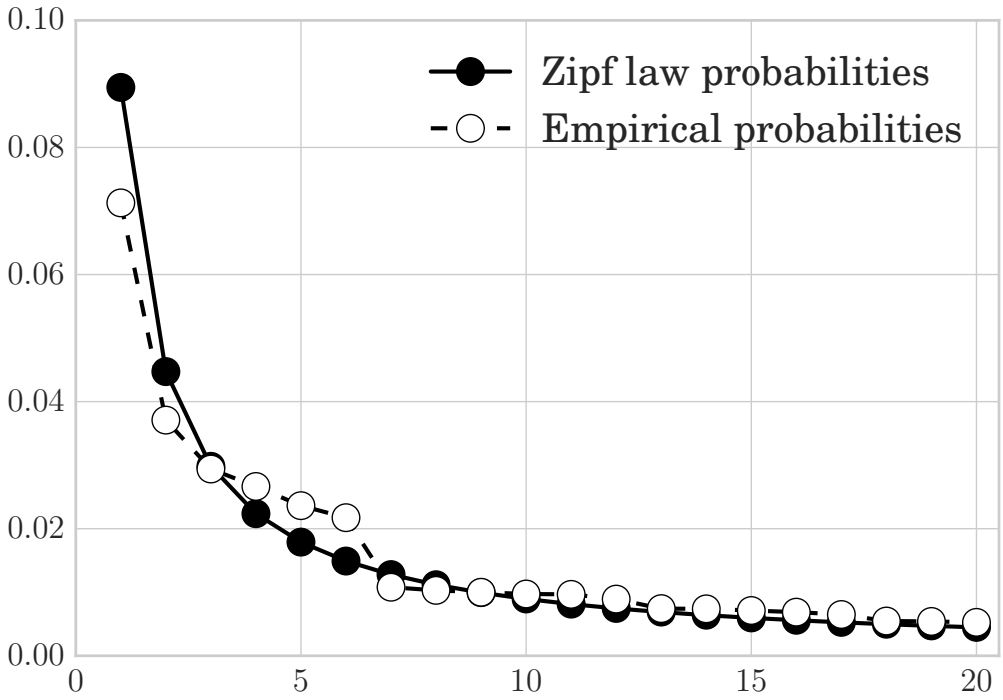
$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots$$

# 20 Most Common Words in the Brown Corpus

| Rank | Word | Empirical Probability | Zipf's Law |
|---|---|---|---|
| 1 | THE | 0.0712741770532 | 0.0695687332667 |
| 2 | OF | 0.03709015642 | 0.0347843666334 |
| 3 | AND | 0.0293903735907 | 0.0231895777556 |
| 4 | TO | 0.0266451804799 | 0.0173921833167 |
| 5 | A | 0.0236269959948 | 0.0139137466533 |
| 6 | IN | 0.0217343916163 | 0.0115947888778 |
| 7 | THAT | 0.0107913082806 | 0.00993839046668 |
| 8 | IS | 0.0102972753831 | 0.00869609165834 |
| 9 | WAS | 0.00999779977101 | 0.00772985925186 |
| 10 | HE | 0.00972582702126 | 0.00695687332667 |
| 11 | FOR | 0.00966572817393 | 0.00632443029698 |
| 12 | IT | 0.00892315089089 | 0.0057973944389 |
| 13 | WITH | 0.0074247542059 | 0.00535144102052 |
| 14 | AS | 0.00738808372279 | 0.00496919523334 |
| 15 | HIS | 0.00712629721834 | 0.00463791555112 |
| 16 | ON | 0.00686654796295 | 0.00434804582917 |
| 17 | BE | 0.0064957686337 | 0.00409227842746 |
| 18 | AT | 0.00547205098012 | 0.00386492962593 |
| 19 | BY | 0.00540482176108 | 0.0036615122772 |
| 20 | I | 0.00526017707769 | 0.00347843666334 |

The Brown corpus contains 981,716 words, 40,234 unique ones.

# Zipf's Distribution

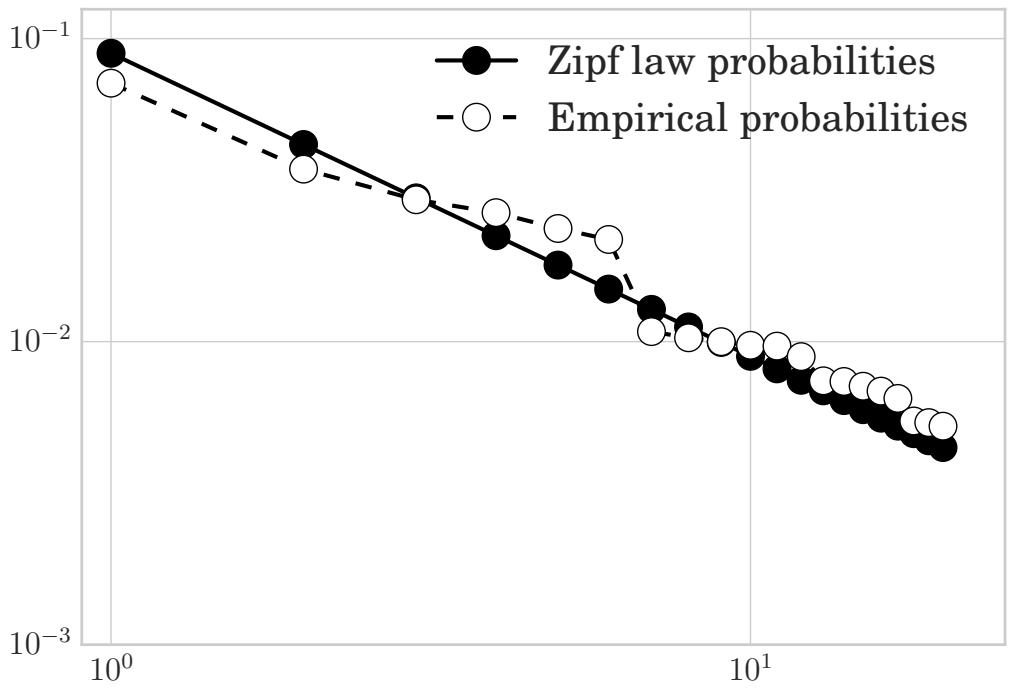



The graph shows the Zipf distribution and the empirical distribution for the first 20 words of the Brown corpus.

# Conversion to Logarithmic Scale

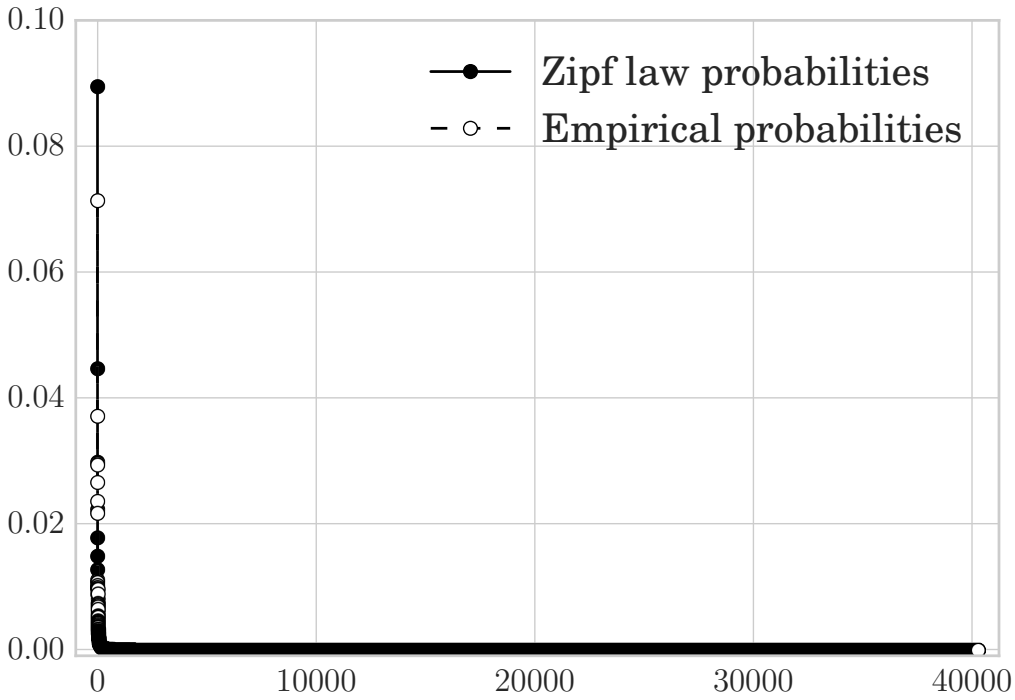
- When we see a function that declines rapidly, we can try to see its graph in logarithmic scale.
- Instead of $y = f(x)$, we plot $\log y = \log f(x)$.
- Similarly, we take the logarithm of $x$ in the horizontal axis.

# Zipf's Distribution in Logarithmic Axes



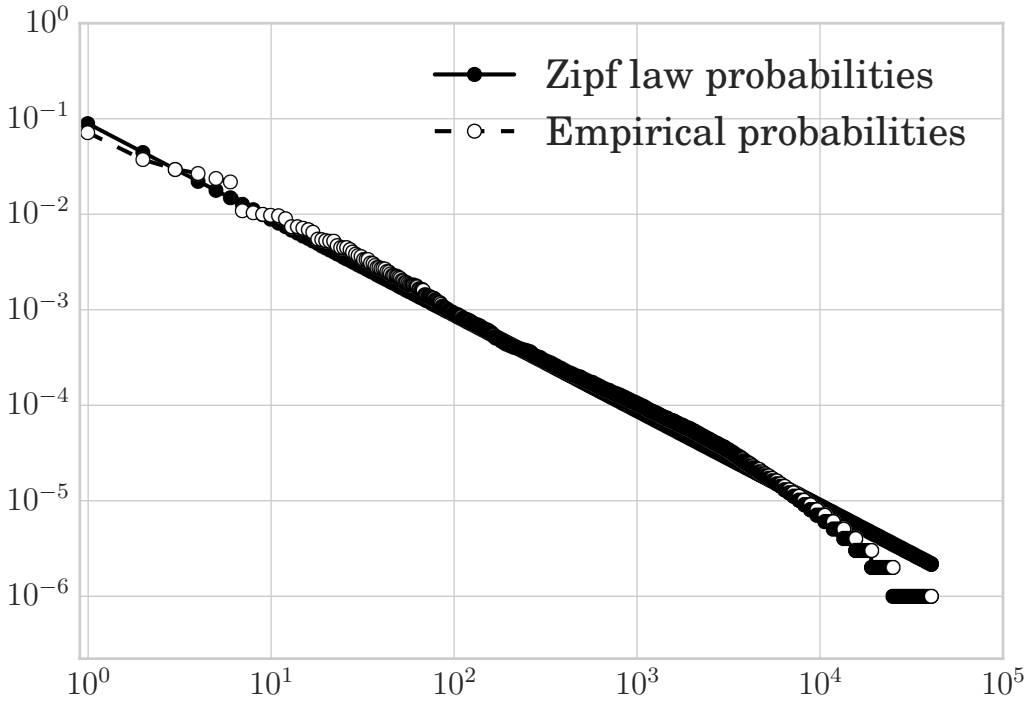

The graph shows Zipf's distribution and the empirical distribution for the first 20 words of the Brown corpus in logarithmic axes.

# Graph of the Full Zipf's Distribution




The graph shows Zipf's distribution and the empirical distribution for all words of the Brown corpus.

# Graph of the Full Zipf's Distribution in Logarithmic Axes




The graph shows Zipf's distribution and the empirical distribution for all words of the Brown corpus in logarithmic axes.

# Power Laws

- Zipf's law is an example of a *power law*.
- A power law occurs when the probability of a value is proportional to a negative power of that value, or in mathematical terms:

$$P(X = x) \propto cx^{-k} : c > 0,\ k > 0$$

# Explanation of the Form of Logarithmic Graphs

If we have:

$$y = cx^{-k}$$

then we get:

$$\log y = \log(cx^{-k}) = \log c - k \log x$$

This is a straight line that cuts the $y$ axis at $\log c$ and has a slope equal to $-k$. When our data show up in logarithmic axes as a straight line, it is probable that we have a distribution that follows a power law.

# The Pareto Principle

- According to the *Pareto Principle*, about 20% of the causes are responsible for about 80% of the effects.
- In popular folklore, 20% of the people do about 80% of the work.
- The mathematical formulation is:

$$P(X = x) = c/x^{1-\theta}$$

where

$$\theta = \log .80/\log .20$$

# Where Do We Find Power Laws?

Where do we find power laws? In addition to the previous examples:

- Citations.
- Earthquake magnitudes.
- The diameter of moon craters.
- The increase in numbers of biological species in time.
- Fractals.
- Foraging patterns of predators.
- Peak gamma-ray intensity of solar flares.
- Number of long distance calls in a day.
- Number of people affected by blackouts.
- Occurrence of family names.
- ...

# Benford's Law

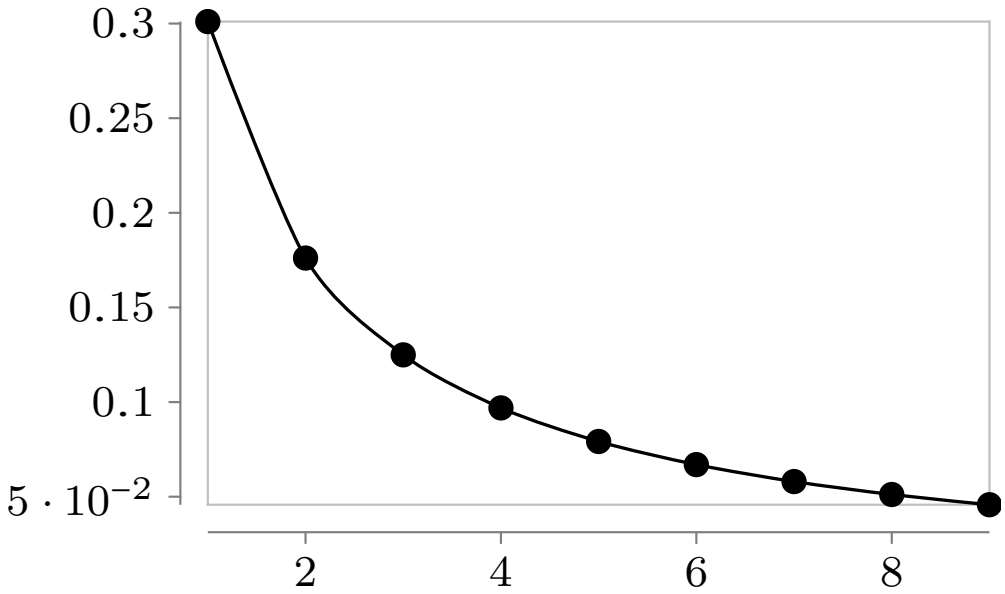
- Benford's law was proposed by the physicist Frank Benford in 1938. It had also been proposed previously by Simon Newcomb in 1881.
- It refers to the frequency distribution of digits in many kinds of data.
- The first digit of a number is 1 in 30% of the time. It is 2–9 with decreasing probability.
- Specifically, the law states that the probability that the first digit of a number is $d = 1, 2, \ldots, 9$ is equal to:

$$P(d) = \log\left(1 + \frac{1}{d}\right)$$

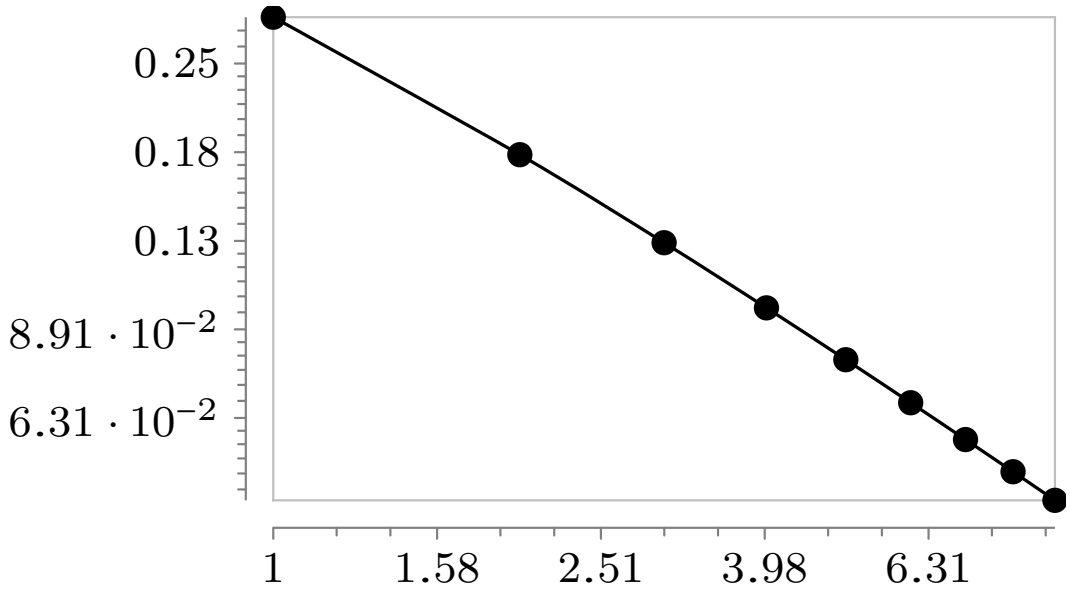
# Benford's Law Probabilities

| Leading Digit | Probability |
|---|---|
| 1 | 0.301029995664 |
| 2 | 0.176091259056 |
| 3 | 0.124938736608 |
| 4 | 0.0969100130081 |
| 5 | 0.0791812460476 |
| 6 | 0.0669467896306 |
| 7 | 0.0579919469777 |
| 8 | 0.0511525224474 |
| 9 | 0.0457574905607 |

# Benford's Law Graph



# Benford's Law in Logarithmic Axes



# Where Do We Encounter Benford's Law?

Where do we encounter Benford's Law? About everywhere:

- Physical constants.
- Heights of highest buildings.
- Populations.
- Stock prices.
- Addresses.
- ...

In fact, if some data do not follow Benford's law, it may be an indication that they have been doctored.

# Benford's Law and Sequential Search

- If the keys of the data are numeric and they follow Benford's law, and
- the keys we search for also follow Benford's law, then:
- there will be more records and searches starting with 1, then with 2, and so on.

# Outline


1. Sequential Search
2. The Matthew Effect and Power Laws
3. Self-Organizing Search
4. The Secretary Problem
5. Binary Search
6. Representing Integers in Computers
7. Binary Search Revisited
8. Comparison Trees

# Self-Organizing Search

- In *self-organizing search* we try to take advantage of the popularity of keys and records.
- The basic idea is to bring the most popular records forward, so that we can find them faster than the less popular ones.

# Self-Organizing Search with Move-To-Front

- In the *move-to-front* method, we move to the beginning of our elements each record that we find.

# Algorithm of Self-Organizing Search with Move-To-Front

**Algorithm:** Self-organizing search with move-to-front.

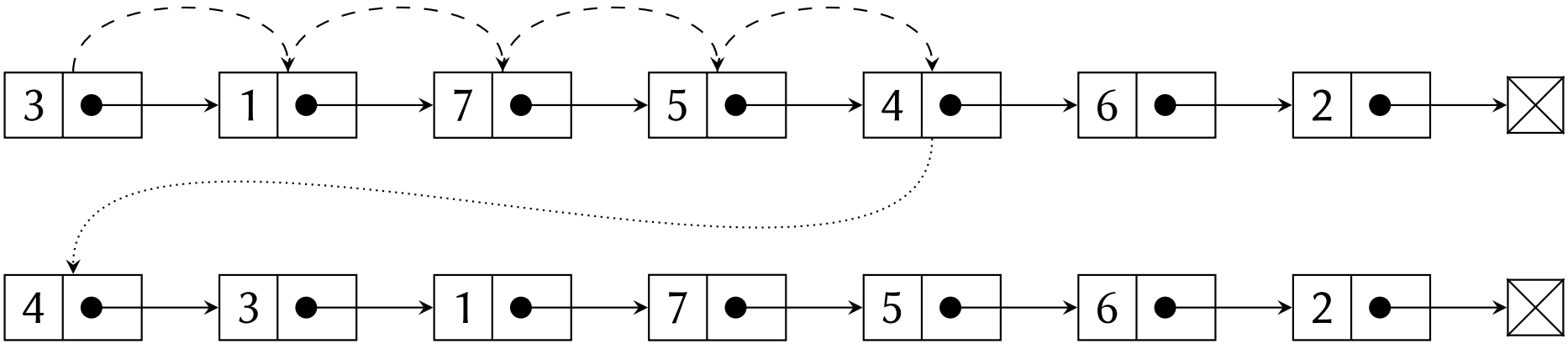
MoveToFrontSearch($L$, $s$) $\longrightarrow$ $s$ or NULL

**Input:** $L$, a list of items
$s$, an element we are searching for

**Output:** the element we are looking for, or NULL if it does not exist

```
1  p ← NULL
2  foreach r in L do
3      if Matches(r, s) then
4          if p ≠ NULL then
5              m ← RemoveListNode(L, p, r)
6              InsertListNode(L, NULL, m)
7              return m
8          return r
9      p ← r
10 return NULL
```

# Move-to-Front Examples



# Algorithm Requirements

- The algorithm requires moving an element at the beginning of our elements, which is easy to do if our elements are in a list, instead of an array.
- The `RemoveListNode`($L$, $p$, $r$) function removes item $r$ from the list $L$ and returns it; $p$ is the item before $r$ in the list.
- The `InsertListNode`($L$, NULL, $m$) function inserts element $m$ in the first position of list $L$.

# The Transposition Method

- In the *transposition method* we swap each method that we find with the record that comes before it.

# Search-Organizing Search with Transposition

**Algorithm:** Self-organizing search with transposition.

`TranspositionSearch`$(L, s) \longrightarrow s$ or NULL

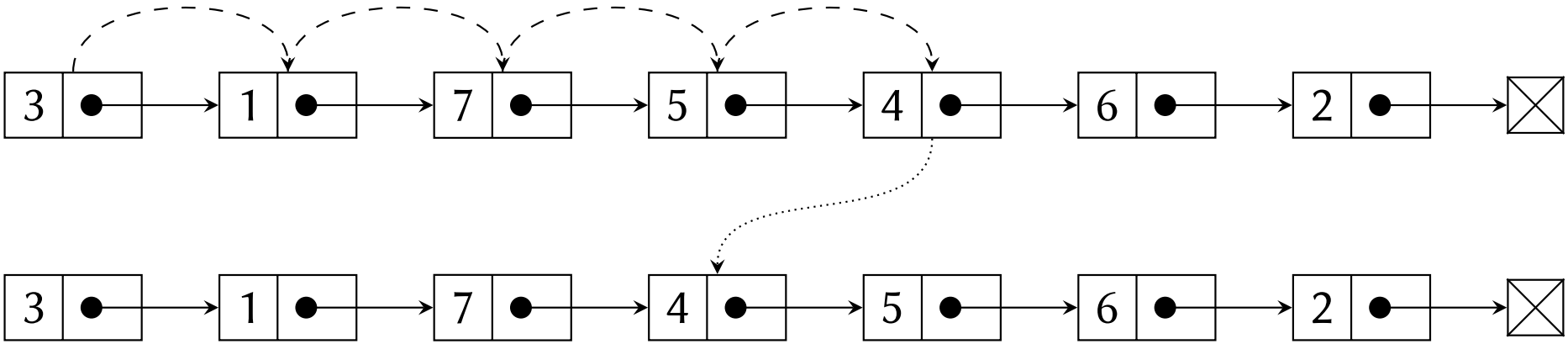
**Input:** $L$, a list of items

$s$, an element we are searching for

**Output:** the element we are looking for, or NULL if it does not exist

```
1  p ← NULL
2  q ← NULL
3  foreach r in L do
4      if Matches(r, s) then
5          if p ≠ NULL then
6              m ← RemoveListNode(L, p, r)
7              InsertListNode(L, q, m)
8              return m
9          return r
10     q ← p
11     p ← r
12 return NULL
```

# Transposition Method Example



# Algorithm Requirements

- The implementation must be able to insert an element not just in the beginning of a list, but in any position.
- Therefore we use the `InsertListNode`($L$, $q$, $m$) function, where $q$ is the element preceding $m$.

# The Transposition Method in Arrays

- Swapping the positions of two elements is something that we can do easily in an array; that means that the transposition method can be used with arrays, as well as with lists.

# Self-Organizing Search with Transposition in an Array

**Algorithm:** Self-organizing search with the transposition method in an array.

TranspositionArraySearch$(A, s) \rightarrow i$

**Input:** $A$, an array of items
$s$, an element we are searching for

**Output:** $i$, the position of $s$ in $A$ if $A$ contains $s$, or $-1$ otherwise

```
1  for i ← 0 to |A| do
2      if Matches(A[i], s) then
3          if i > 0 then
4              Swap(A[i − 1], A[i])
5              return i − 1
6          else
7              return i
8  return −1
```

# Self-Organizing Search Performance

- It can be proved that the average number of comparisons that we need in the move-to-front method is $O(n/\lg n)$, much better than simple sequential search.
- The transposition method requires even fewer comparisons, but that happens in the long run.
- Therefore the best option is the move-to-front method, except if we know that we will be performing many searches.

# Outline


1. Sequential Search
2. The Matthew Effect and Power Laws
3. Self-Organizing Search
4. The Secretary Problem
5. Binary Search
6. Representing Integers in Computers
7. Binary Search Revisited
8. Comparison Trees

# The General Setting

- Suppose that we are looking for the best among a set of elements (whatever that means).
- However, each time we examine an item we must decide *on the spot* if it is the right one.
- If we reject it, we cannot recall it later.
- If we choose it, then we will ignore any items further on.

# The Secretary Problem

- Suppose we want to hire a secretary.
- We have a pile of CVs in front of us.
- We call each candidate in turn.
- If it's the right candidate, we hire the candidate on the spot.
- If not, we reject the candidate and we move on to the next.
- We cannot call back a candidate that we have already rejected.

What is the best way to select the candidate?

# Solution of the Secretary Problem

- If we have $n$ CVs, we examine the first $n/e$, where $e \approx 2.7182$ is Euler's number.
- We reject the first $n/e$ candidates, keeping note of the best one among them. Therefore we reject the first $37\%$ of candidates.
- We choose the first of the remaining candidates that is better than the one we have noted, if such a candidate exists. Otherwise we do not pick any.
- It can be proved that the probability of picking the best candidate is $1/e$, i.e., $37\%$.

# Secretary Search Algorithm

**Algorithm:** Secretary problem.

SecretarySearch$(A, s) \rightarrow i$

**Input:** $A$, an array of items

**Output:** $i$, the position of the element in $A$ that we should choose, or $-1$ if we failed to find the best element

```
1  m ← ⌈|A|/e⌉
2  c ← 0
3  for i ← 1 to m do
4      if Compare(A[i], A[c]) > 0 then
5          c ← i
6  for i ← m to |A| do
7      if Compare(A[i], A[c]) > 0 then
8          return i
9  return −1
```

# The Compare Function

- We use the Compare function to compare one element with another.
- The call Compare$(a, b)$ returns $+1$ if $a$ is better than $b$, $-1$ if $a$ is worse than $b$, and $0$ if they are equal.
- Such comparison functions are typical in programming languages.

# Features of the Secretary Algorithm

- The secretary algorithm will not always give us the best solution.
- It will find the best solution in 37% of the cases.
- There is no algorithm that can find the best solution in most cases.

# Optimal Stopping

- The secretary problem is an instance of *optimal stopping*, which deals with choosing when to take an action to maximize a reward or minimize a cost.
- Such problems are handled by *online algorithms* where input is presented and processed in a serial function, piece by piece, without having the entire input available to us from the start.
- *Streaming algorithms*, where the input comes as a sequence of items that can be examined in a single pass as they come, are examples of online algorithms.

# Secretary Search Algorithm

**Algorithm:** Secretary problem.

SecretarySearch$(A, s) \rightarrow i$

**Input:** $A$, an array of items

**Output:** $i$, the position of the element in $A$ that we should choose, or $-1$ if we failed to find the best element

```
1  m ← ⌈|A|/e⌉
2  c ← 0
3  for i ← 1 to m do
4      if Compare(A[i], A[c]) > 0 then
5          c ← i
6  for i ← m to |A| do
7      if Compare(A[i], A[c]) > 0 then
8          return i
9  return −1
```

# Performance of the Secretary Algorithm

- In lines 1–5 we go through the first $m$ elements
- Lines 6–8 are similar to lines 1–3 of sequential search, but instead of $n$ items we have $n - m$ items.
- Therefore the overall complexity is
  $O(m + (n - m + 1)/2) = O(m/2 + (n + 1)/2 = O(n/2e + (n - 1)/2) = O(n)$.

# A Variation of the Secretary Problem

- In the secretary problem we are 100% happy with the best choice and 0% with any other choice.
- But we may not be that absolute.
- Our satisfaction with a candidate may correspond to the candidate's expected performance.
- So we will be most satisfied with the best candidate, somewhat less satisfied with the second best, and so on.
- How should we choose the candidate then?

# Solution of the Variation

- It can be proved that we must work as previously, but instead of the first $n/e$ candidates, we must reject, keeping note of the best among them, the first $\sqrt{n}$.
- In that case, the probability that we will find the best candidate goes to 100% as $n$ increases.

# Outline


1. Sequential Search
2. The Matthew Effect and Power Laws
3. Self-Organizing Search
4. The Secretary Problem
5. Binary Search
6. Representing Integers in Computers
7. Binary Search Revisited
8. Comparison Trees

# Search in Ordered Data

- Up to this point, we have assumed that the elements in which we search are not ordered.
- If, however, the elements are ordered, then we will work differently.

# Ordering by Key

- In order to take advantage of ordered data, we must make sure that the data are ordered by the same key as the key we use for searching.

# The First 10 Chemical Elements Order by Atomic Number

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1** 1.0079 | **2** 4.0025 | **3** 6.941 | **4** 9.0122 | **5** 10.811 | **6** 12.011 | **7** 14.007 | **8** 15.999 | **9** 18.998 | **10** 20.180 |
| **H** | **He** | **Li** | **Be** | **B** | **C** | **N** | **O** | **F** | **Ne** |
| Hydrogen | Helium | Lithium | Beryllium | Boron | Carbon | Nitrogen | Oxygen | Fluorine | Neon |

We can take advantage of the ordering if we search by *atomic number*.

# The First 10 Chemical Elements Ordered by Name

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **89** 227 | **13** 26.982 | **95** 243 | **51** 121.76 | **18** 39.948 | **33** 74.922 | **85** 210 | **56** 137.33 | **97** 247 | **4** 9.0122 |
| **Ac** | **Al** | **Am** | **Sb** | **Ar** | **As** | **At** | **Ba** | **Bk** | **Be** |
| Actinium | Aluminum | Americium | Antimony | Argon | Arsenic | Astatine | Barium | Berkelium | Beryllium |

We can take advantage of the ordering if we search by *name*.

# The First 10 Chemical Elements Ordered by Symbol

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **89** 227 | **47** 107.87 | **13** 26.982 | **95** 243 | **18** 39.948 | **33** 74.922 | **85** 210 | **79** 196.97 | **5** 10.811 | **56** 137.33 |
| **Ac** | **Ag** | **Al** | **Am** | **Ar** | **As** | **At** | **Au** | **B** | **Ba** |
| Actinium | Silver | Aluminum | Americium | Argon | Arsenic | Astatine | Gold | Boron | Barium |

We can take advantage of the ordering if we search by *chemical symbol*.

# Taking Advantage of Order

- If we have a pile of CVs ordered alphabetically based on surnames, we can immediately take advantage of the ordering.
- If we are looking for a candidate whose surname starts with “D”, we will start searching near the top of the pile.
- If we are looking for a candidate whose surname starts with “T”, we will start searching near the end of the pile.

# Guess a Number

- Somebody asks "I have in mind a number between 0 and 100, can you guess which one with the fewest guesses? For each of your guesses I will tell you if you got the number right, or if the number is greater than or lower than your guess."
- Unless you are psychic, the winning strategy is to start by taking the middle number, 50.
- If the number is greater than 50, then the next guess will be 75, half of the interval from 50 to 100.
- If the number is smaller than 50, then the next guess will be 25, half of the interval from 0 to 50.

# Binary Search

- That is the idea behind the *binary search* algorithm.
- Each time we try to find an item, if we do not land on it, we divide the search space by two.

# Binary Search Algorithm

**Algorithm:** Binary search.

`BinarySearch`$(A, s) \rightarrow i$

**Input:** $A$, a sorted array of items
$s$, an element we are searching for

**Output:** $i$, the position of $s$ in $A$ if $A$ contains $s$, or $-1$ otherwise

```
1  l ← 0
2  h ← |A|
3  while l ≤ h do
4      m ← ⌊(l + h)/2⌋
5      c ← Compare(A[m], s)
6      if c < 0 then
7          l ← m + 1
8      else if c > 0 then
9          h ← m − 1
10     else
11         return m
12 return −1
```

# Operation of Binary Search

- We use three variables $l$, $h$, $m$.
- The variable $l$ refers to the lower end of the search space.
- The variable $h$ refers to the upper end of the search space.
- The variable $m$ refers to the middle of the search space.
- When we start, $l$ is the position of the first element and $h$ is the position of the last element.

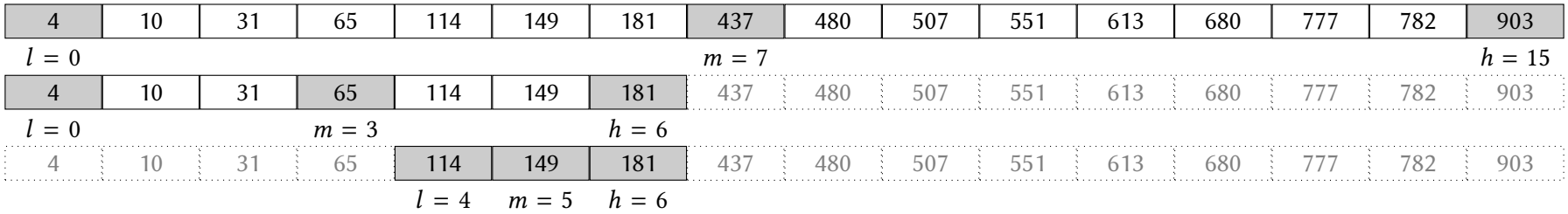
# Binary Search Termination

- The binary search algorithm always terminates.
- If it finds the element, it returns it.
- Otherwise, each time round the loop we reduce the search space by half.
- That cannot go on for ever, so if it does not find the element the algorithm will still terminate and return $-1$.

# Binary Search Termination (Contd.)

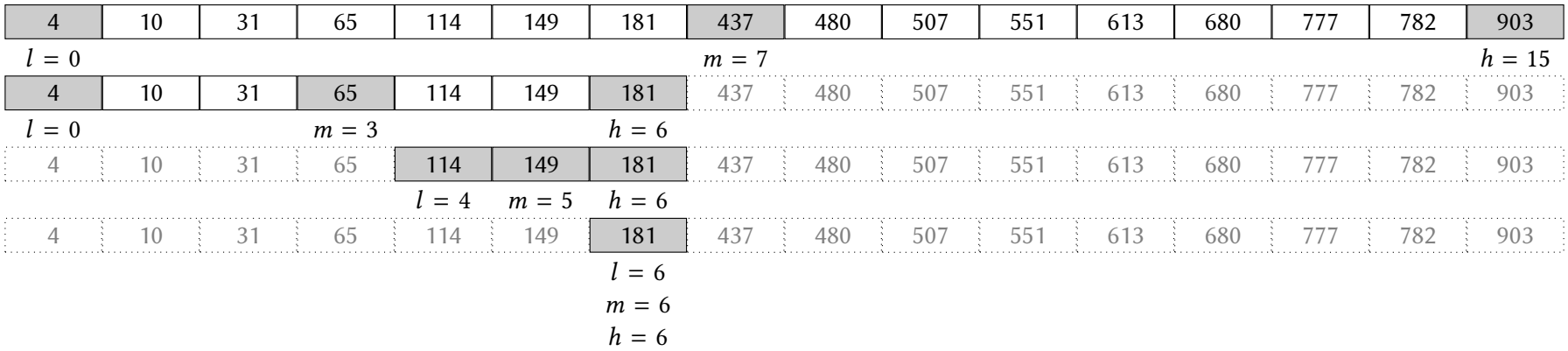
- The loop is repeated as long as $l \leq h$.
- In each iteration $l$ increases or $h$ decreases.
- At some point the repeat condition will cease to hold.

# Successful Search (1)



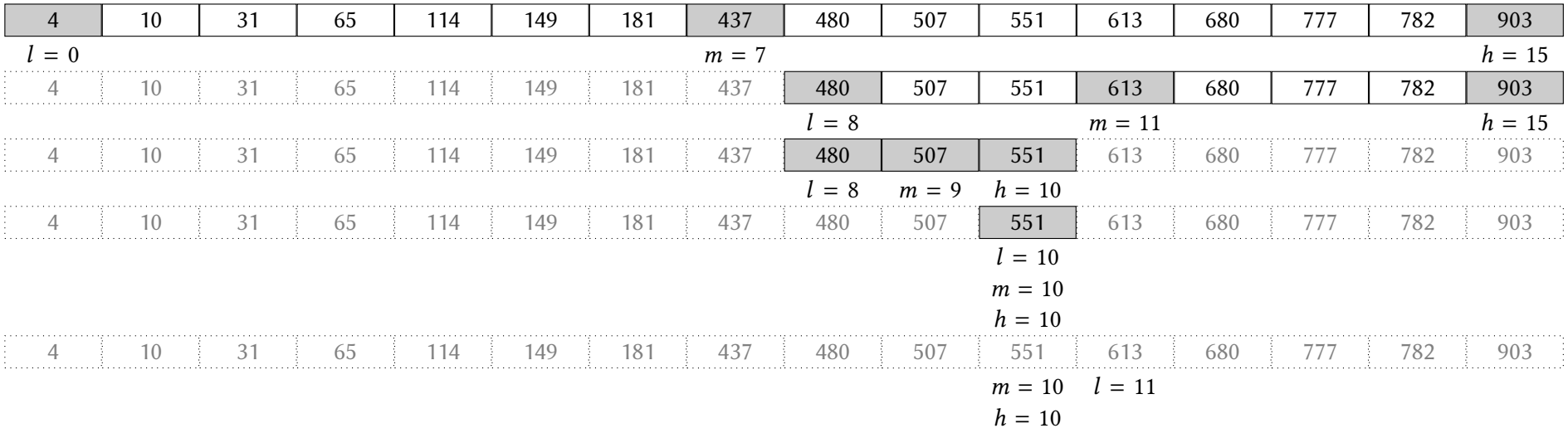

Successful search for 149.

# Successful Search (2)



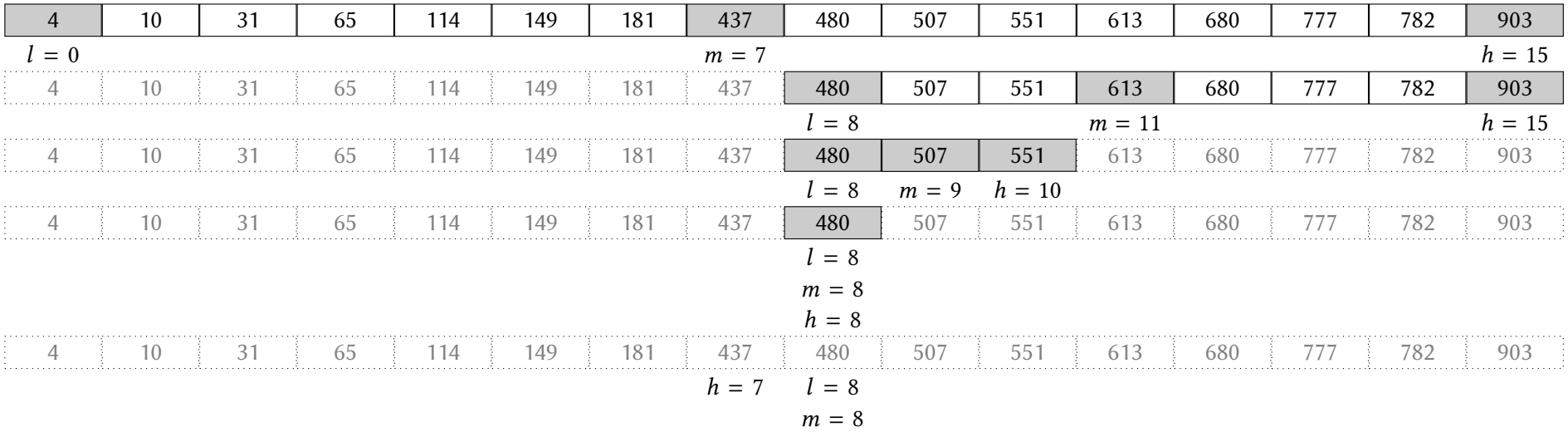

Successful search for 181.

# Unsuccessful Search (1)




Unsuccessful search for 583.

# Unsuccessful Search (2)




Unsuccessful search for 450.

# History of Binary Search

- John Mauchly, one of the designers of ENIAC, the first general purpose digital computer, described binary search in 1946. It was not clear what should be done if the items were not a power of two.
- The first full description was given in 1960, by Derrick Henry Lehmer, a mathematician and computer pioneer.

# Problems with Binary Search

- The story of binary search is full of problems.
- An prominent programmer and researcher, Jon Bentley, found in the 1980s that about 90% of professional programmers could not implement it correctly, even after working several hours on it.
- Another researcher in 1988 found that only five out of 20 textbooks had accurate descriptions of the algorithm.
- Jon Bentley's implementation was also wrong, and the error remained hidden for about 20 years!
- It was discovered by Joshua Bloch, an acclaimed software engineer who implemented many features of the Java programming language.

# The Nature of Programming

> *It was on one of my journeys between the EDSAC room and the punching equipment that 'hesitating at the angles of stairs' the realisation came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.*

Maurice Wilkes, another computer pioneer, reminiscing about the early computing days of 1949, when he was programming the EDSAC computer in Cambridge.

# Binary Search Algorithm

**Algorithm:** Binary search.

`BinarySearch`$(A, s) \rightarrow i$

**Input:** $A$, a sorted array of items
$s$, an element we are searching for

**Output:** $i$, the position of $s$ in $A$ if $A$ contains $s$, or $-1$ otherwise

```
1  l ← 0
2  h ← |A|
3  while l ≤ h do
4      m ← ⌊(l + h)/2⌋
5      c ← Compare(A[m], s)
6      if c < 0 then
7          l ← m + 1
8      else if c > 0 then
9          h ← m − 1
10     else
11         return m
12 return −1
```

# Where is the Problem?

- The algorithm is correct.
- The programs that implement it can be wrong.
- The error is in line 4.
- When we add two positive numbers we get a bigger positive number.
- Except if the addition is carried out by a computer...

# Example of Problematic Addition

- In many programming languages we can see things like:

$$1.500.500.000 + 1.500.000.000 = -1.294.967.296$$

Why?

# Outline


1. Sequential Search
2. The Matthew Effect and Power Laws
3. Self-Organizing Search
4. The Secretary Problem
5. Binary Search
6. Representing Integers in Computers
7. Binary Search Revisited
8. Comparison Trees

# How Many Integers Can We Represent?

- To represent integers we usually represent a fixed number of bits, e.g., 32 bits or 64 bits.
- If a number consists of $n$ bits, $B_{n-1} \ldots B_1 B_0$, its value is:

$$B_{n-1} \times 2^{n-1} + B_{n-2} \times 2^{n-2} + \cdots + B_1 \times 2^1 + B_0 \times 2^0$$

- If all bits are 0, the number is equal to 0.
- If all bits are 1, the number is equal to:

$$2^{n-1} + 2^{n-2} + \cdots + 2^1 + 2^0$$

# Four Bits Binary Numbers (1)

- With four bits, when all bits are 0 we get the binary number $0000 = 0$, whereas when all bits are 1 we get the binary number 1111, which is equal to:

$$2^3 + 2^2 + 2^1 + 2^0$$

- That is one less than the binary number 10000, which is equal to: $2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 2^4$.

# Four Bits Binary Numbers (2)

- We will use $(b)_2$ to show that $b$ is a binary number, e.g., $(10)_2$, and not a decimal number.
- So we have:

$$(10000)_2 = (1111)_2 + 1$$

- Therefore:

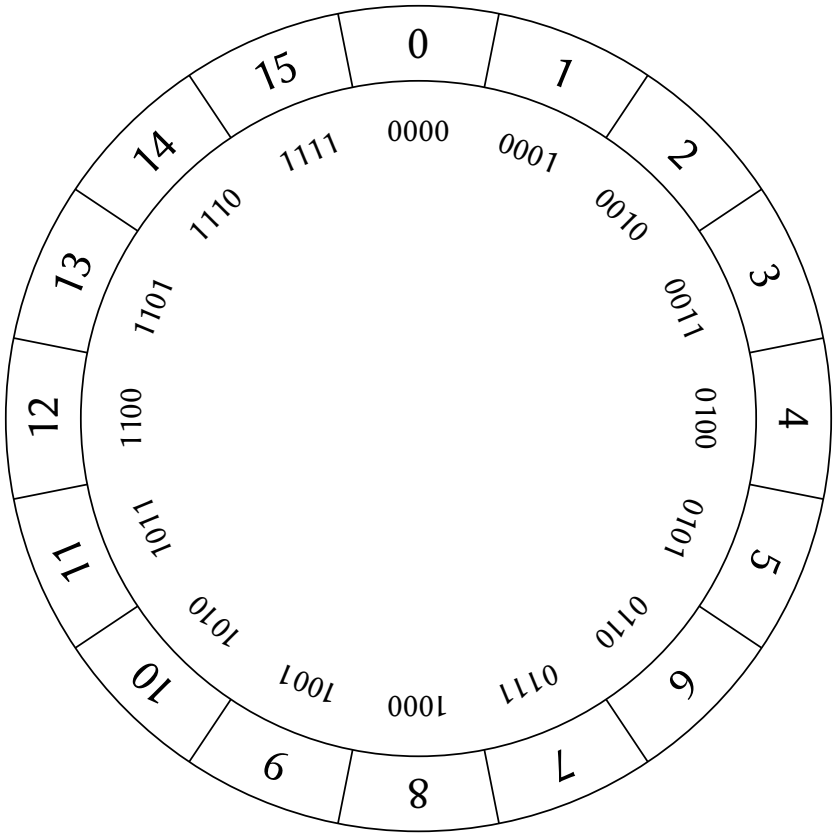
$$(1111)_2 = (10000)_2 - 1 = 2^4 - 1$$

- Which means that with four bits we can represent the positive numbers from 0 up to $2^4 - 1$, which are $2^4$ in total.

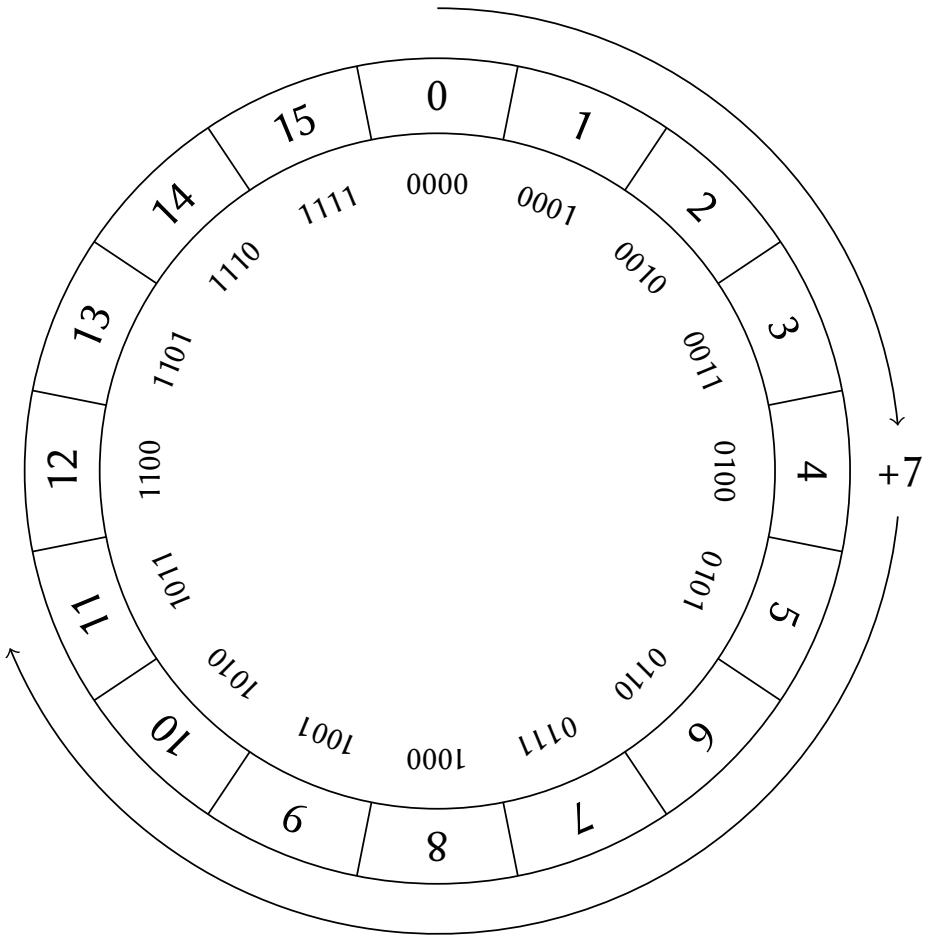
# Binary Numbers with $n$ Digits

- We will use $d\{k\}$ to show that the digit $d$ is repeated $k$ times.
- In this way we will write $1\{4\}$ instead of $1111$.
- With $n$ bits we can represent all numbers from $0$ to $(1\{n\})_2$, which is equal to:
  $2^{n-1} + 2^{n-2} + \cdots + 2^1 + 2^0$
- That is one less than:
  $(10\{n\})_2$: $(10\{n\})_2 = (1\{n\})_2 + 1$.
- So $(1\{n\})_2 = (10\{n\})_2 - 1$, and as $(10\{n\})_2 = 2^n$, we get $(1\{n\})_2 = 2^n - 1$.
- Therefore with $n$ bits we can represent all numbers from $0$ to $2^n - 1$, $2^n$ numbers in total.

# Numbers Wheel

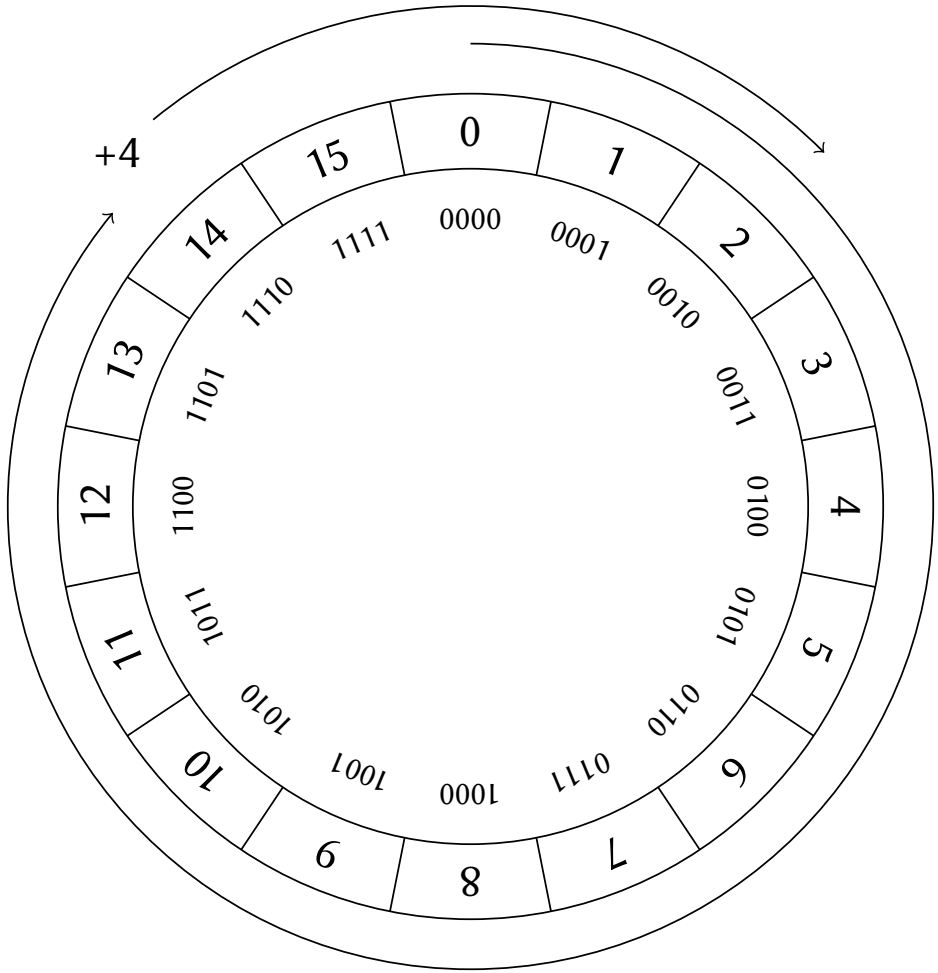


# Addition in the Numbers Wheel




Calculating $4 + 7 = 11$.

# Addition with Overflow




Calculating $14 + 4 = 2$ because of *overflow*.

# Overflow and Binary Search

- We calculate $m \leftarrow \lfloor (l + h)/2 \rfloor$
- If the sum of $l$ and $h$ is greater than the biggest integer, we will have overflow and the algorithm will not work correctly.
- If we use 32 bits to represent positive integers, we can represent the numbers from 0 to $2^{32} - 1$, or from 0 to 4,294,967,295.
- If the numbers sum up to more than 4.29 billion we will have overflow.
- That might have seen astronomical some years back, but data sets of such sizes are pretty common now.

# Representing Signed Integers

- To this point we have been dealing with positive numbers only.
- What happens if we want to represent negative numbers as well?
- In that case we keep the first bit to show the sign of the number; we call it the *sign bit*.
- If the first bit is 0, the number is positive. If the first bit is 1, the number is negative.
- That means that we have $n - 1$ bits to represent the values of the numbers. So we have $2^{n-1}$ positive numbers and $2^{n-1}$ negative numbers (we include 0 in the positive numbers).
- Half of the wheel will be positive and half will be negative. Therefore, if we add two big positive numbers, we may end up with a negative!

# Two Complement's Representation

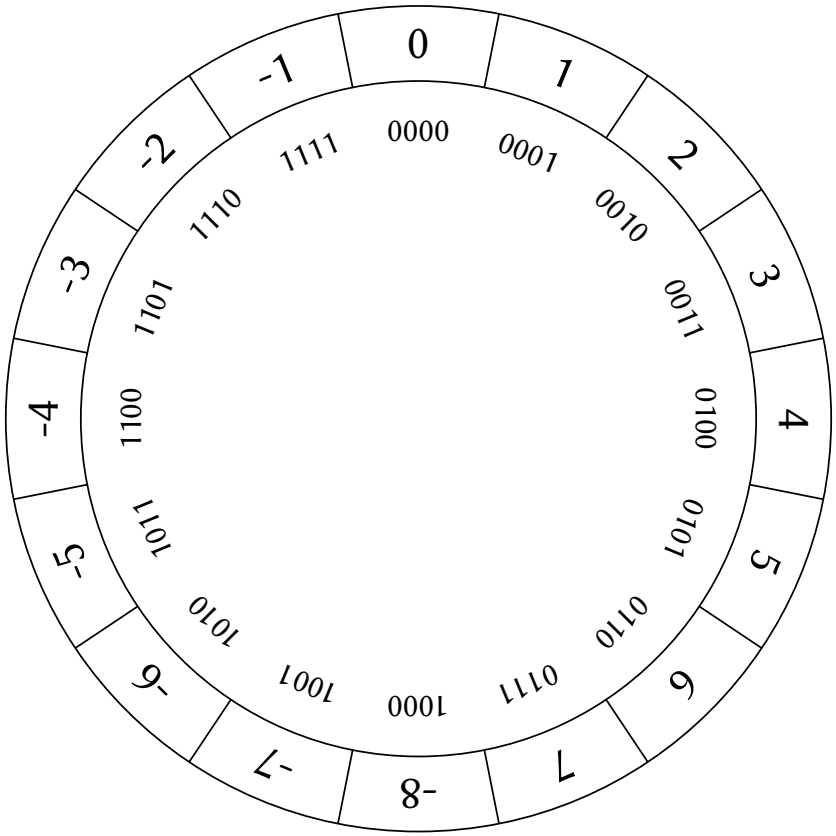
- In *two complement's representation*, the negative of a positive number $x$ of $n$ bits is represented by the binary number:

$$c = 2^n - x$$

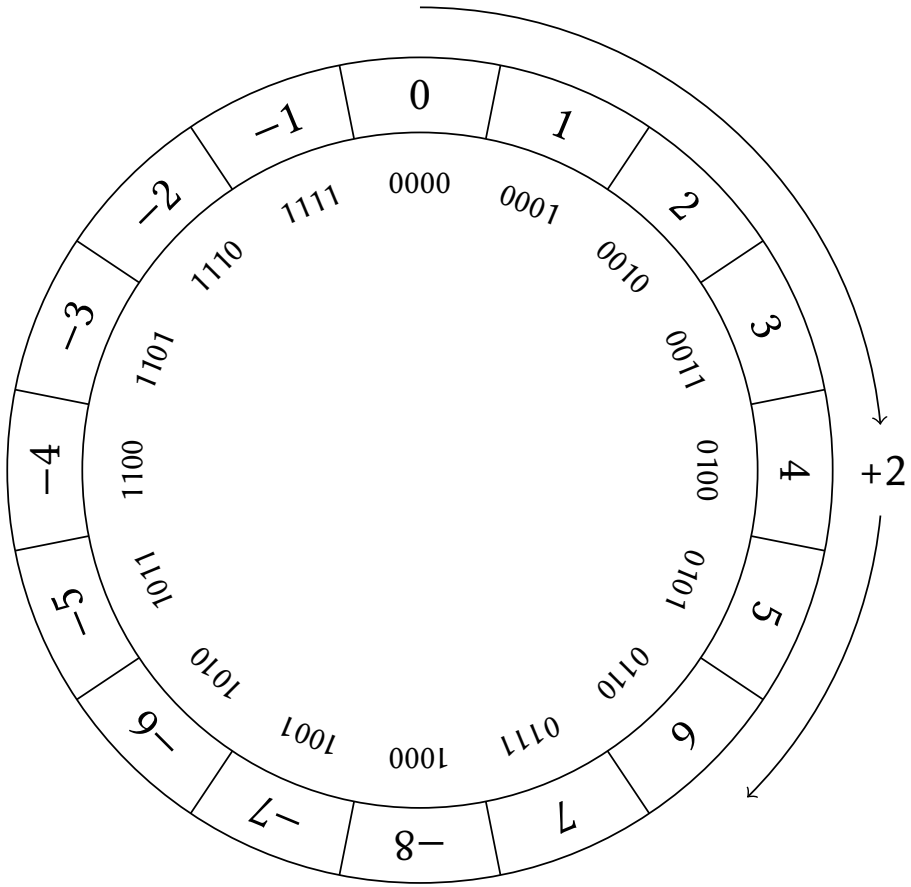
- That is, the negative of $x$ is the number that added to $x$ is equal to $2^n$.
- That explains the name, as we are complementing with a power of 2.
- For example, the opposite of $5 = (0101)_2$ will be

$$2^4 - 5 = 16 - 5 = 11 = (1011)_2$$

# Two Complement's Numbers Wheel

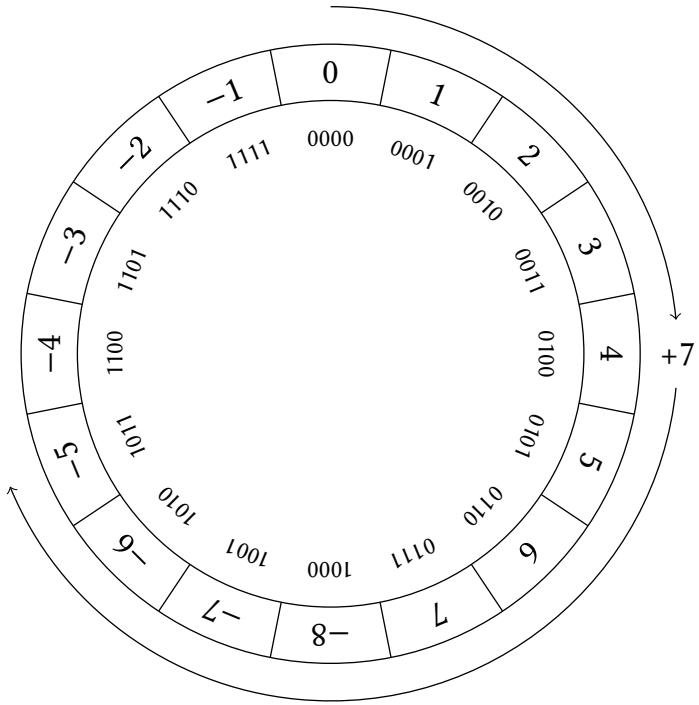


# Adding with Two Complement's Representation




Calculating $4 + 2 = 6$ in two complement's representation.

# Overflow in Two Complement's Representation




Calculating $4 + 7 = -5$ with two complement's representation, results in overflow.

# Overflow and Two Complement's Representation

- If we have $n$ digits, as we have $2^{n-1}$ numbers for the positives (and 0) and $2^{n-1}$ numbers for the negatives, we can represent the numbers from 0 to $2^{n-1} - 1$ and from $-1$ to $-2^{n-1}$.
- With 32 bits, the biggest positive numbers is 2,147,483,647.
- If we go beyond that we will get to $-2^{32} = -2{,}147{,}483{,}648$.
- This is what humanity did in 2014 by watching the video of the popular song "Gangnam Style." On December 1, we learned that the number of views of that video on YouTube had exceeded 2,147,483,647.
- YouTube was updated and now uses a 64 bits counter. This will overflow after $2^{63} - 1 = 9{,}223{,}372{,}036{,}854{,}775{,}807$ views.

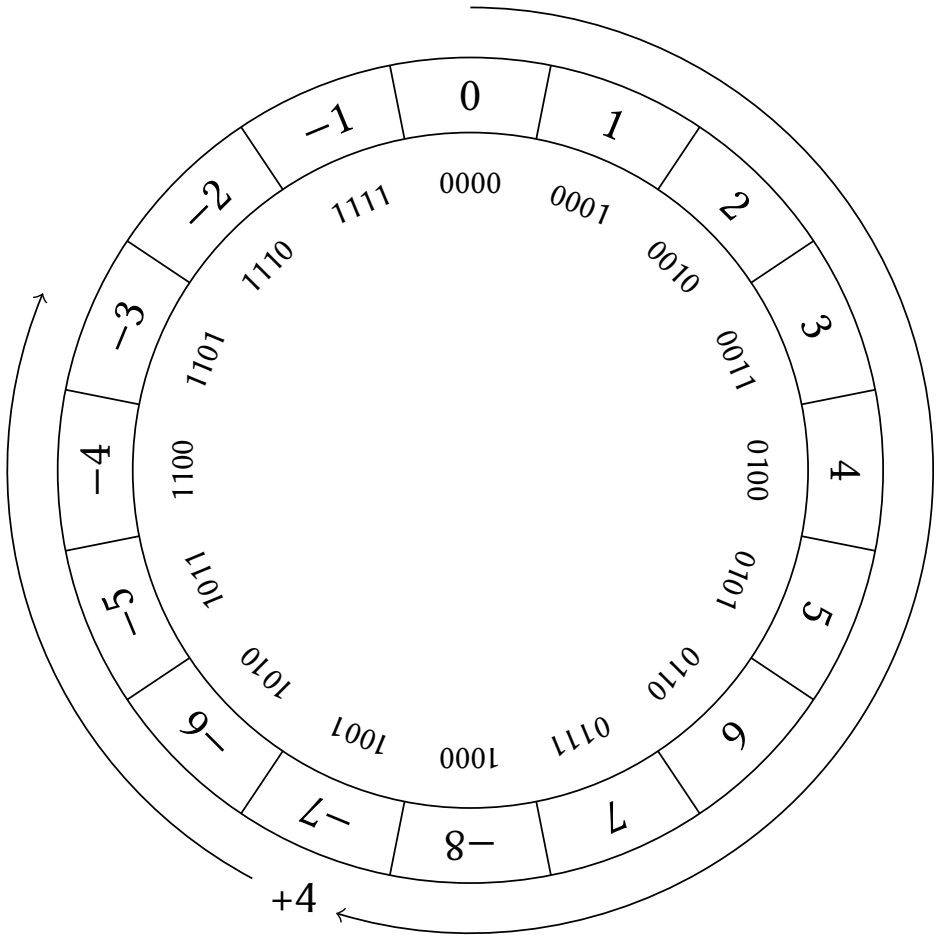
# Why Do We Use Two Complement's Representation?

- A seemingly easier representation would be flipping all bits from 0 to 1 and from 1 to 0.
- For example, $(0010)_2 = 2$ and $-2$ would be $(1101)_2$.
- This is called *ones' complement representation*, because we complement each digit with 1. That is why it is called "ones' " and not "one's".
- Another option would be to create the opposite of a number by just flipping the sign bit.
- For example, $(0010)_2 = 2$ and $-2$ would be $(1010)_2$.
- This is called *signed magnitude representation*.

# Advantages of Two Complement's Representation

- The two alternatives that we mentioned have two representations for 0: $(0000)_2$, $(1111)_2$ in ones' complement representation and $(0000)_2$, $(1000)_2$ in signed magnitude representation.
- Also, addition and subtraction are easier with two complement's representation.

# Subtraction with Two Complement's Representation




Calculating $4 - 7 = -3$ with two complement's representation.

# Outline


1. Sequential Search
2. The Matthew Effect and Power Laws
3. Self-Organizing Search
4. The Secretary Problem
5. Binary Search
6. Representing Integers in Computers
7. Binary Search Revisited
8. Comparison Trees

# Fixing Binary Search

Instead of calculating:

$$\lfloor (l + h)/2 \rfloor$$

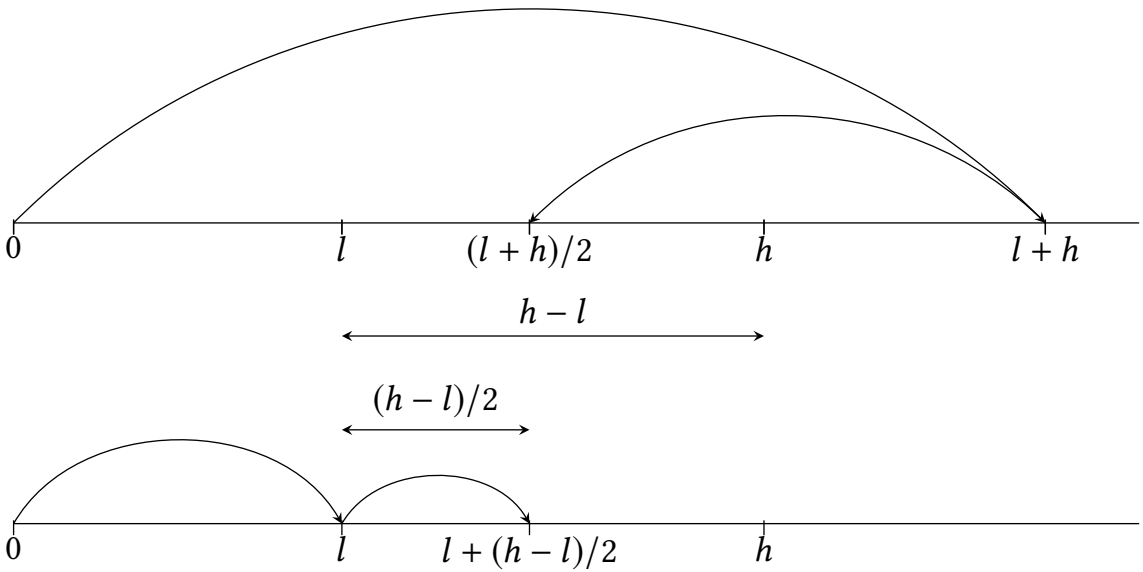
we calculate the equivalent:

$$l + \lfloor (h - l)/2 \rfloor$$

Indeed, as $l$ is integer, we have:

$$l + \lfloor (h - l)/2 \rfloor = \lfloor l + (h - l)/2 \rfloor = \lfloor (l + h)/2 \rfloor$$

# Fix Diagram



# Binary Search without Overflow Algorithm

**Algorithm:** Binary search without overflows.

`SafeBinarySearch`$(A, s) \rightarrow i$

**Input:** $A$, a sorted array of items
$s$, an element we are looking for

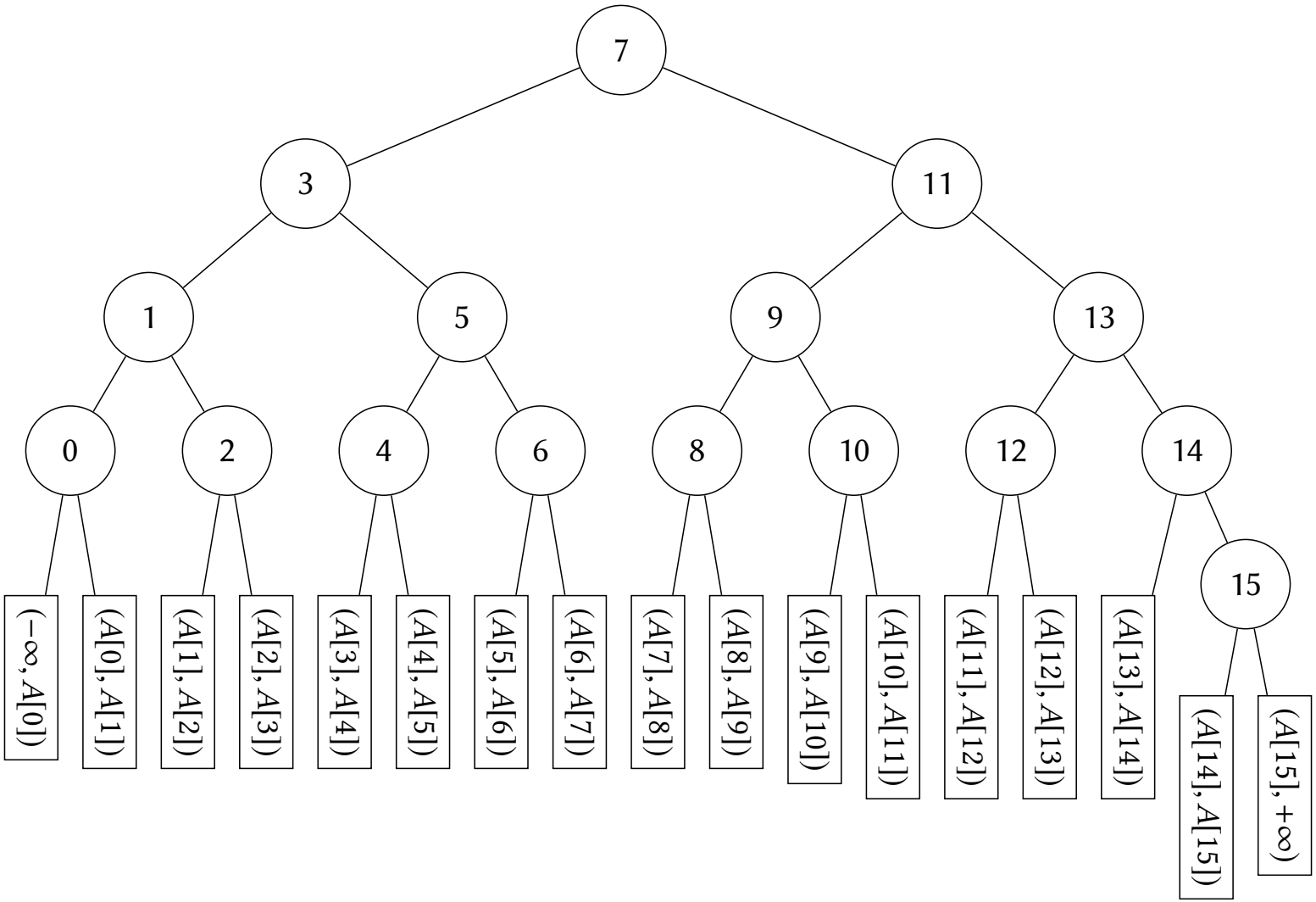
**Output:** $i$, the position of $s$ in $A$ if $A$ contains $s$, or $-1$ otherwise

```
1  l ← 0
2  h ← |A|
3  while l ≤ h do
4      m ← l + ⌊(h − l)/2⌋
5      c ←Compare(A[m], s)
6      if c < 0 then
7          l ← m + 1
8      else if c > 0 then
9          h ← m − 1
10     else
11         return m
12 return −1
```

# Outline


1. Sequential Search
2. The Matthew Effect and Power Laws
3. Self-Organizing Search
4. The Secretary Problem
5. Binary Search
6. Representing Integers in Computers
7. Binary Search Revisited
8. Comparison Trees

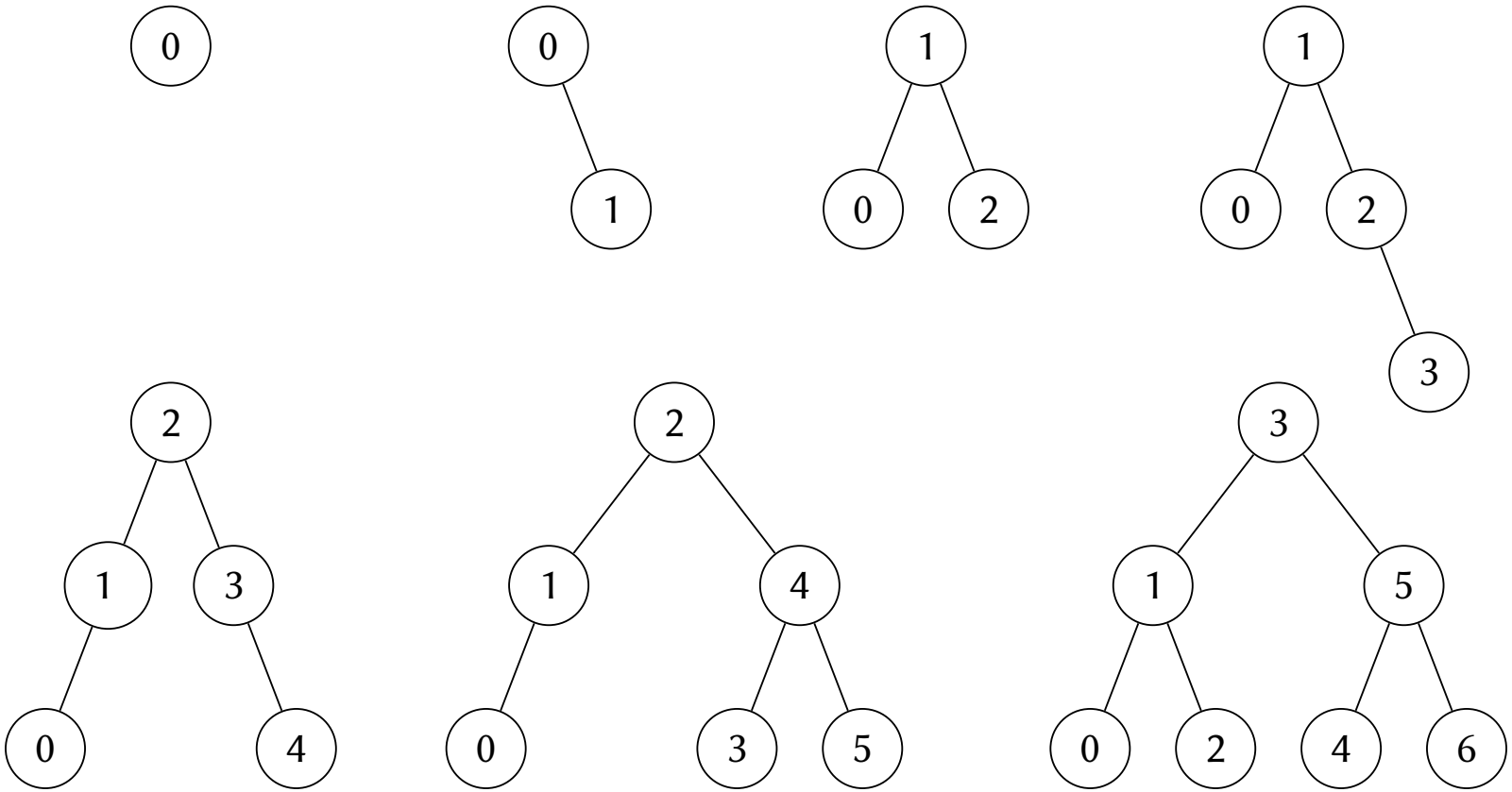
# Comparison Tree for 16 Items



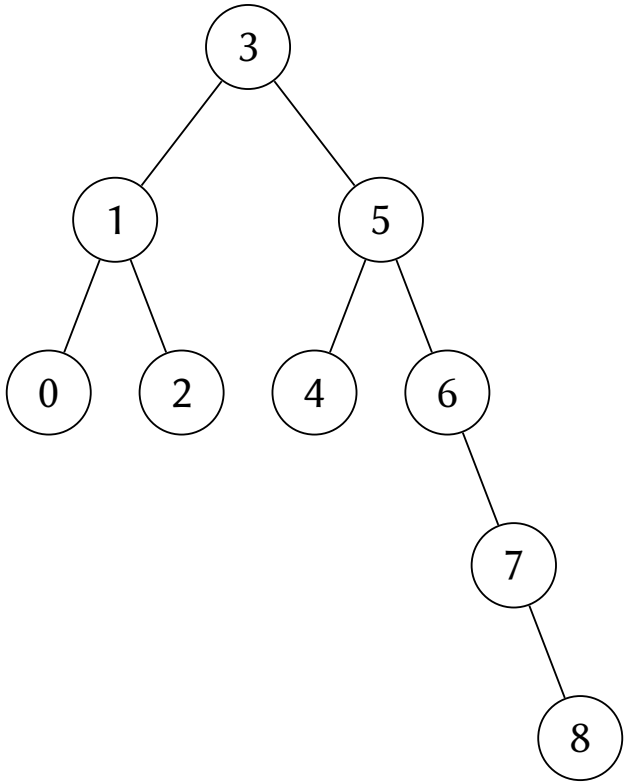
# Comparison Tree Construction

- If we have $n$ elements in array $A$, we take as root of the tree the $A[\lfloor n/2 \rfloor]$ element.
- We put the first $\lfloor n/2 \rfloor - 1$ elements in the left subtree and the left $n - \lfloor n/2 \rfloor$ in the right subtree.
- We proceed in this way in each subtree, until we arrive at subtrees that have only one node.

# Comparison Trees of 1 to 7 Items



# Infeasible Comparison Tree



# Comparison Tree Analysis

- Level zero has 1 node.
- Level one has up to 2 nodes.
- Level two can have up to $2 \times 2 = 2^2$ nodes.
- Level $k$ can have up to $2^k$ nodes.
- As $1 = 2^0$ and $2 = 2^1$, a tree with $k$ levels can have up to $2^0 + 2^1 + \cdots + 2^k$ nodes.
- That is equal to $(1\{k+1\})_2 = 2^{k+1} - 1$ nodes.
- So, if $n = |A|$, we have $n \leq 2^{k+1} - 1$, or $n < 2^{k+1}$.
- Similarly, all levels before the $k$th level are full, so we have $n \geq 2^k$.

# Binary Search Analysis

- A successful search can stop at any level and requires from 1 to $k + 1$ comparisons (we start from level 0), so if $2^k \leq n < 2^{k+1}$, the comparisons will be from 1 to $k + 1$.
- Regarding unsuccessful searches, if $n = 2^{k+1} - 1$ the last level is full, so we need exactly $k + 1$ comparisons.
- If $2^k \leq n < 2^{k+1} - 1$ the last level is not full, so we need $k$ or $k + 1$ comparisons.

# Binary Search Complexity

- A successful search requires $O(\lg n)$ time.
- An unsuccessful search requires $\Theta(\lg n)$ time.

# Comparison Examples

- For 100 items, as in the number guessing game, we will need $\lg 100 \approx 6.65 < 7$ guesses.
- We have $2^{32} - 1 = 4{,}294{,}967{,}295$. So any search in 4,294,967,295 sorted elements will take no more than 32 comparisons.

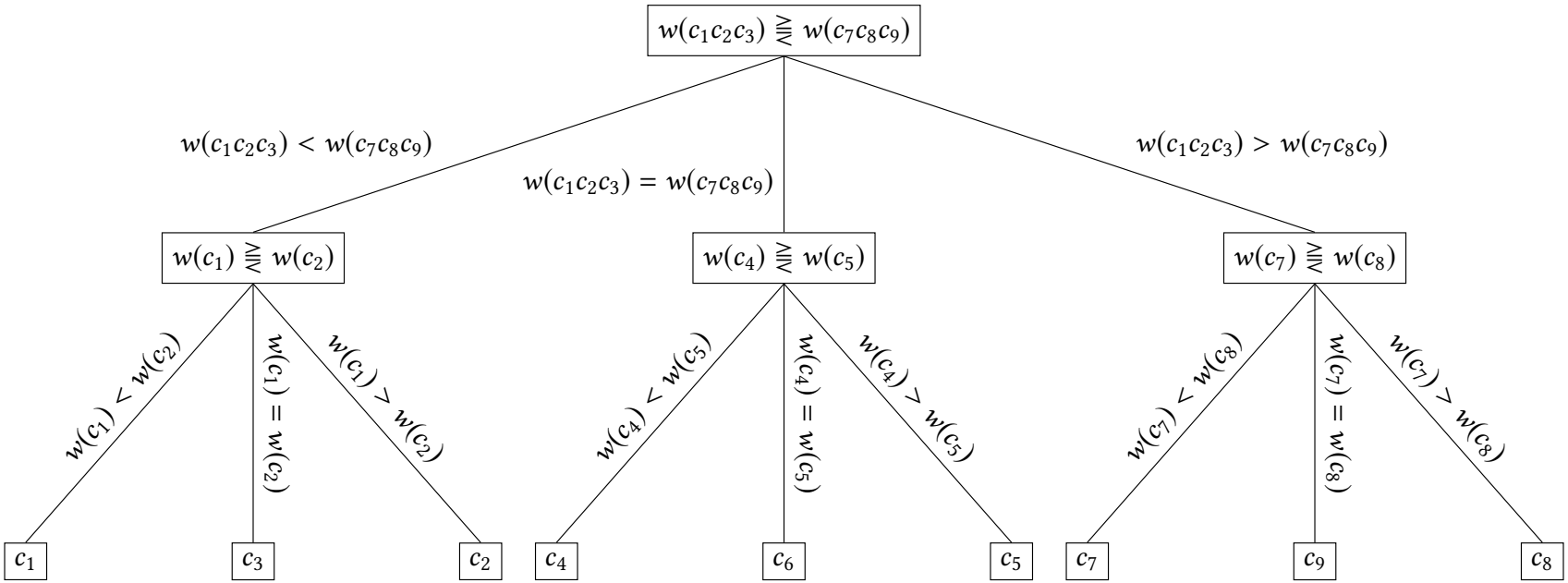
# Exponential Growth

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $2^{56}$ | $2^{57}$ | $2^{58}$ | $2^{59}$ | $2^{60}$ | $2^{61}$ | $2^{62}$ | $2^{63}$ |
| $2^{48}$ | $2^{49}$ | $2^{50}$ | $2^{51}$ | $2^{52}$ | $2^{53}$ | $2^{54}$ | $2^{55}$ |
| $2^{40}$ | $2^{41}$ | $2^{42}$ | $2^{43}$ | $2^{44}$ | $2^{45}$ | $2^{46}$ | $2^{47}$ |
| $2^{32}$ | $2^{33}$ | $2^{34}$ | $2^{35}$ | $2^{36}$ | $2^{37}$ | $2^{38}$ | $2^{39}$ |
| $2^{24}$ | $2^{25}$ | $2^{26}$ | $2^{27}$ | $2^{28}$ | $2^{29}$ | $2^{30}$ | $2^{31}$ |
| $2^{16}$ | $2^{17}$ | $2^{18}$ | $2^{19}$ | $2^{20}$ | $2^{21}$ | $2^{22}$ | $2^{23}$ |
| $2^{8}$ | $2^{9}$ | $2^{10}$ | $2^{11}$ | $2^{12}$ | $2^{13}$ | $2^{14}$ | $2^{15}$ |
| $2^{0}$ | $2^{1}$ | $2^{2}$ | $2^{3}$ | $2^{4}$ | $2^{5}$ | $2^{6}$ | $2^{7}$ |

# The Coin Weighing Problem

- We have nine coins $c_1, c_2, \ldots, c_9$.
- One of them is counterfeit and has a different weight than the rest.
- We have one balance.
- How can we find the counterfeit with two weighings?

# Coin Weighing Tree




The height of the comparison tree is $\log_3(9) = 2$.