Discrete Structures: Algorithms

Amotz Bar-Noy

Department of Computer and Information Science
Brooklyn College

Outline

Algorithm: Definitions

- A finite set of precise instructions for performing a computation or for solving a problem.
- A specific set of instructions for carrying out a procedure or solving a problem, usually with the requirement that the procedure terminates at some point.
- A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation.
- A step-by-step procedure for solving a problem or accomplishing some end especially by a computer.
- A logical arithmetical or computational procedure that if correctly applied ensures the solution of a problem.
- A finite set of unambiguous instructions performed in a prescribed sequence to achieve a goal, especially a mathematical rule or procedure used to compute a desired result.

Algorithm: Definitions

- A word used by programmers when they do not want to explain what they did.

- A word used by those whose program failed to justify what they did.

Algorithm

Synonym

- Method, Procedure, Program, Code, Process, Recipe, Prescription, Routine, Solution, Technique, Mechanism, Scheme, Way, Design, Plan, Strategy, Construction, . . .

Etymology

- Alteration of Middle English algorisme;
- from Old French & Medieval Latin algorismus;
- from Arabic al-khuwarizmi;
- from the name of the 9^{th}-century Persian Mathematician Al-Khowârizmi who was the first (???) to formalize the rules for the four basic arithmetic operations.

The Ultimate Algorithmic Problem!?

Question

- What is required to **solve** problems and/or design **efficient** algorithms?

Attributes

1. Talent?
2. Intuition?
3. Luck?
4. Experience?
5. Hard work?

Answer

- Apply some combination of these five attributes!!!

Some Heuristics to Solve Problems

1. Search for a pattern
2. Draw a figure
3. Formulate an equivalent problem
4. Modify the problem
5. Choose effective notation
6. Exploit symmetry
7. Divide into cases
8. Work backward
9. Argue by contradiction
10. Pursue parity
11. Consider extreme cases
12. Generalize

Algorithms: Online Videos

What is an algorithm?

- https://www.youtube.com/watch?v=Da5TOXCwLSg
- https://www.youtube.com/watch?v=6hfOvs8pY1k
- https://www.youtube.com/watch?v=CvSOaYi89B4&feature=youtu.be

Why algorithms are called algorithms?

- https://www.youtube.com/watch?v=oRkNaF0QvnI

Three Ancient Algorithms

The Babylonian Multiplication Algorithm

- Introduced around **3700** years ago

The Euclid's Greatest Common Divisor Algorithm

- Introduced around **2300** years ago

The Sieve of Eratosthenes to Find Prime Numbers Algorithm

- Introduced around **2200** years ago

The Babylonian Multiplication Algorithm

- Although there are some evidences of early multiplication algorithms in Egypt (around 1700-2000 BC) the oldest algorithm is widely accepted to have been found on a set of Babylonian clay tablets that date to around 1600-1800 BC.
- Their true significance only came to light in 1972 when computer scientist & mathematician Donald E. Knuth published the first English translations of various Cuneiform mathematical tablets.
- The Babylonians had developed a nice way to explain an algorithm by examples as the algorithm itself was being defined.
- The tablets also appear to have been an early form of instruction manual.

The Euclid's Greatest Common Divisor Algorithm

- The Euclidian algorithm is a procedure used to find the greatest common divisors (GCD) of two positive integers.
- It was first described by Euclid in his manuscript the Elements written around 300 BC.
- It is a very efficient computation that is still used today by computers in some form or other.

The Sieve of Eratosthenes Algorithm

- The Sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to any given limit.
- It is attributed to the Greek mathematician Eratosthenes of Cyrene and was “invented” around 200 BC.
- The algorithm iteratively marks as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2.
- The “less efficient” method sequentially tests each candidate number for divisibility by previously found prime.

Algorithms — Properties

Correctness

- For all valid inputs

Termination

- Does not run forever on some inputs

Complexity – Efficiency

- As a function of the input size
- Worst-Case and/or Average-Case

Scalability

- “Similar” structure and efficiency for any input size

Limitations

- For the algorithm and for the problem

Optimality

- Optimal or near-optimal or approximately optimal solutions

Cost and Complexity

Cost

- How much resources does the algorithm require?
  - Usually time and space (memory)

Complexity

- As a function of the input size
  - Usually an integer $n > 0$
  - Usually a monotonic non-decreasing function

Terminology

- Complexity is often called **running-time** because time is the dominating cost

Worst Case and Average Case Complexity

Worst case (informal definition)

- $T(n)$ is the **worst case complexity** if among **all** inputs of size n the worst case complexity is $T(n)$

Average case (informal definition)

- $T(n)$ is the **average case complexity** if the **average** complexity over **all** length n inputs is $T(n)$ where averaging is based on some distribution of the inputs, usually the uniform distribution

Bounds (informal definition)

- A function $f(n)$ is an **upper bound** if $T(n) \leq f(n)$ for all n
- A function $g(n)$ is a **lower bound** if $T(n) \geq g(n)$ for all n
- A function $h(n)$ is a **tight bound** if $T(n) \approx h(n)$ for all n

Performance Evaluation of Algorithms

Theoretical analysis

- **All** possible inputs
- Independent of hardware/software implementation
- Based on a **high level** language

Experimental Study

- Some **typical** input
- Depends on hardware/software implementation
- Based on a **real** program

Growth of Functions

Objective

- Develop a language to express that Algorithm A is better than or worse than or equivalent to Algorithm B

Technique

- Define a "$\leq$" relation between functions measuring the **growth** of functions

Robustness

- Being independent of the hardware/software environment: Turing machines, classroom models, today computers, and super-computers

An important property

- Constants that can be affected by changing the environment should be ignored

Examples of Function Growth

Running Time	1 second	1 minute	1 hour
n	1,000,000	60,000,000	3,600,000,000
n^2	1,000	7,745	60,000
n^4	31	88	244
2^n	19	25	31

- Maximum size of a problem that can be solved in one second, one minute, and one hour, for various running times measured in **microseconds**

Examples of Function Growth

Running Time	New Maximum Size
n	$256m$
n^2	$16m$
n^4	$4m$
2^n	$m+8$

- Increase in the maximum size of a problem that can be solved with a certain complexity, by using a computer that is **256 times faster** than the previous one.
- Each entry is given as a function of m, the previous maximum problem size

Examples of Function Growth

Running Time	1 second	1 minute	1 hour
$n/256$	256,000,000	15,360,000,000	9,216,600,000,000
$n^2/256 = (n/16)^2$	16,000	123,925	960,000
$n^4/256 = (n/4)^4$	126	352	979
$2^n/256 = 2^{n-8}$	27	33	39

- Maximum size of a problem that can be solved by the **faster** computer in one second, one minute, and one hour, for various running times measured in **microseconds**

The "O, Ω, Θ, o, ω" Notation

Settings

Let f and g be positive functions on the non-negative integers

Big-Oh

$f(n) = O(g(n))$ if $f(n)$ is asymptotically less than or equal to $g(n)$

Big-Omega

$f(n) = \Omega(g(n))$ if $f(n)$ is asymptotically greater than or equal to $g(n)$

Big-Theta

$f(n) = \Theta(g(n))$ if $f(n)$ is asymptotically equal to $g(n)$

Little-oh

$f(n) = o(g(n))$ if $f(n)$ is asymptotically strictly less than $g(n)$

Little-omega

$f(n) = \omega(g(n))$ if $f(n)$ is asymptotically strictly greater than $g(n)$

Big-Oh, Big-Omega, and Big-Theta

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

- **There exist** a real constant $c > 0$ and an integer constant $n_0 > 0$ such that $f(n) \leq cg(n)$ **for every** integer $n \geq n_0$

$f(n) = \Omega(g(n))$

- **There exist** a real constant $c > 0$ and an integer constant $n_0 > 0$ such that $f(n) \geq cg(n)$ **for every** integer $n \geq n_0$

$f(n) = \Theta(g(n))$

- **There exist** two real constants $c', c'' > 0$ and an integer constant $n_0 > 0$ such that $c''g(n) \leq f(n) \leq c'g(n)$ **for every** integer $n \geq n_0$

Big-Oh, Big-Omega, and Big-Theta

Notational abuses

- Assume $\Psi \in \{O, \Omega, \Theta\}$
- $\Psi(g(n))$ is a set of functions.
- $f_1(n) = \Psi(g(n))$ and $f_2(n) = \Psi(g(n))$ do not imply $f_1(n) = f_2(n)$
- $f(n) \in \Psi(g(n))$ is the more accurate notation
- $f(n) = \Psi(g(n))$ is often written as $f = \Psi(g)$

Big-Oh and Big-Omega

	$f(n) = O(g(n))$	$g(n) = O(f(n))$
$g(n)$ grows faster	YES	NO
$f(n)$ grows faster	NO	YES
same growth	YES	YES

	$f(n) = \Omega(g(n))$	$g(n) = \Omega(f(n))$
$g(n)$ grows faster	NO	YES
$f(n)$ grows faster	YES	NO
same growth	YES	YES

$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

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

- By the definition of O, there exist $c > 0$ and $n_0 > 0$ such that $f(n) \leq cg(n)$ for every $n \geq n_0$
- It follows that $g(n) \geq (1/c)f(n)$ for every $n \geq n_0$
- Since $1/c > 0$, by the definition of Ω, $g(n) = \Omega(f(n))$

Assume $g(n) = \Omega(f(n))$

- By the definition of Ω, there exist $c > 0$ and $n_0 > 0$ such that $g(n) \geq cf(n)$ for every $n \geq n_0$
- It follows that $f(n) \leq (1/c)g(n)$ for every $n \geq n_0$
- Since $1/c > 0$, by the definition of O, $f(n) = O(g(n))$

$f(n) = \Theta(g(n)) \Leftrightarrow$ **(**$f(n) = O(g(n))$ **and** $f(n) = \Omega(g(n))$**)**

Assume $f(n) = \Theta(g(n))$

- By the definition of Θ, there exist $c', c'' > 0$ and $n_0 > 0$ such that $c''g(n) \leq f(n) \leq c'g(n)$ for every $n \geq n_0$
- By the definition of O, $f(n) = O(g(n))$ for $c = c'$ and n_0
- By the definition of Ω, $f(n) = \Omega(g(n))$ for $c = c''$ and n_0

Assume $f(n) = O(g(n))$ **and** $f(n) = \Omega(g(n))$

- By the definition of O, there exist $c_1 > 0$ and $n_1 > 0$ such that $f(n) \leq c_1 g(n)$ for every $n \geq n_1$
- By the definition of Ω, there exist $c_2 > 0$ and $n_2 > 0$ such that $f(n) \geq c_2 g(n)$ for every integer $n \geq n_2$
- Therefore, for $n_0 \geq \max\{n_1, n_2\}$, it follows that $c_2 g(n) \leq f(n) \leq c_1 g(n)$ for every $n \geq n_0$
- By the definition of Θ, $f(n) = \Theta(g(n))$ for $c' = c_1$, $c'' = c_2$, and n_0

O, Ω, Θ as Relations

Θ is an equivalence relation

- **Reflexive:** $f(n) = \Theta(f(n))$
- **Symmetric:** $(f(n) = \Theta(g(n))) \Leftrightarrow (g(n) = \Theta(f(n)))$
- **Transitive:** $f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \Rightarrow (f(n) = \Theta(h(n)))$

O and Ω are reflexive relations

- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$

O and Ω are not symmetric relations

- $f(n) = O(g(n))$ does not imply that $g(n) = O(f(n))$
- $f(n) = \Omega(g(n))$ does not imply that $g(n) = \Omega(f(n))$

O and Ω are transitive relations

- $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- $f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$

n^2 vs. n

$n = O(n^2)$ and $n^2 = \Omega(n)$

- Observe that $n \leq n^2$ for integer $n \geq 1$ ($n < n^2$ for integer $n > 1$)
- Therefore, for $c = 1$ and $n_0 = 1$, the definition of O implies that $n = O(n^2)$ and the definition of Ω implies that $n^2 = \Omega(n)$

$n^2 \neq O(n)$ and $n \neq \Omega(n^2)$

- Observe that if $(1/c) < n$ for a constant $c > 0$, then by multiplying both sides of the inequality by cn, it follows that $n < cn^2$
- Therefore, $n < cn^2$ for every real constant $c > 0$ and integer $n > (1/c)$
- As a result, there are no real constant $c > 0$ and integer n_0 such that $n \geq cn^2$ for every integer $n \geq n_0$
- Consequently, the definitions of O and Ω cannot be applied to get $n^2 = O(n)$ or $n = \Omega(n^2)$

Examples

"Ignore" constants

- $3n = \Theta(n/2)$
- $1000000n = \Theta(n/100000)$
- $\log_2(n) = \Theta(\log_{10}(n))$
- $n \log_2 n/100000 = \Omega(10000000n)$
- $10^{100}n = O(n^2)$

Polynomials

- $a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0 = \Theta(n^d)$
  - for constants $a_0, a_1, \ldots, a_{d-1}$ and a positive constant a_d
- **Example:** $5n^3 + 1000n^2 - 345n + 7 = \Theta(n^3)$

Observations

Eliminating constants

- For any real constant c and $\Psi \in \{O, \Omega, \Theta\}$:
  - $\Psi(cf(n)) = \Psi(f(n))$
  - $\Psi(f(n)/c) = \Psi(f(n))$
  - $\Psi(c) = \Psi(1)$

Addition and multiplication rules

- For $\Psi \in \{O, \Omega, \Theta\}$, if $f_1(n) = \Psi(g_1(n))$ and $f_2(n) = \Psi(g_2(n))$ then
  - $f_1(n) \cdot f_2(n) = \Psi(g_1(n) \cdot g_2(n))$
  - $f_1(n) + f_2(n) = \Psi(g_1(n) + g_2(n)) = \Psi(\max\{g_1(n), g_2(n)\})$

Little-oh and Little-omega

$f(n) = o(g(n))$

- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$:
  - **For any** constant $c > 0$ **there exists** an integer constant $n_0 > 0$ such that $f(n) < cg(n)$ **for every** integer $n \geq n_0$

$f(n) = \omega(g(n))$

- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$:
  - **For any** constant $c > 0$ **there exists** an integer constant $n_0 > 0$ such that $f(n) > cg(n)$ **for every** integer $n \geq n_0$

Propositions

o and ω

- $f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$
- $f(n) = o(g(n)) \wedge g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$
- $f(n) = \omega(g(n)) \wedge g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$

o vs. O

- $f(n) = o(g(n)) \Rightarrow f(n) = O(g(n))$
- $f(n) = O(g(n)) \not\Rightarrow f(n) = o(g(n))$

ω vs Ω

- $f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n))$
- $f(n) = \Omega(g(n)) \not\Rightarrow f(n) = \omega(g(n))$

Examples

Polynomials

- $n^3 = \omega(n^2)$
- $10^{100} n = o(n^2/10^{100})$
- $1 + n + n^2 + n^3 + \cdots + n^{k-1} = o(n^k)$

The logarithmic function

- $\log_2 n = o(n)$
- $n \log_2 n = \omega(n)$

More Examples

The sqrt function

- $\log_2 n = o(\sqrt{n})$
- $n = \omega(\sqrt{n})$

Beyond polynomial function

- $n^k = o(2^n)$ for any integer $k \geq 0$
- $2^n = o(3^n) = o(4^n) = \cdots = o(k^n)$
- $n^n = \omega(n!) = \omega(2^n)$

Hierarchy of Functions

constant	1	
log star	$\log^* n$	
loglog	$\log \log n$	
logarithmic	$\log n$	
poly-logarithmic	$\log^k n$	constant integer $k > 1$
sub-linear	n^ε	constant $0 < \varepsilon < 1$
linear	n	
above-linear	$n \log n$	
quadratic	n^2	
cubic	n^3	
polynomial	n^k	constant integer $k > 3$
super-polynomial	$n^{\log n}$	
exponential	2^n	
factorial	$n!$	
super-exponential	n^n	
exponential tower	$2^{2^{\cdot^{\cdot^{\cdot^2}}}}$	n powers

Analysis of Algorithms

Algorithm $\mathcal{A}$ has a worst case complexity $T(n)$

- To prove that $T(n) = O(f(n))$, show this for **all** size n inputs for **all** $n \geq n_0$ for some integer n_0.
- To prove that $T(n) = \Omega(f(n))$, show this for **one** size n input for **infinitely many** n.
- To prove that $T(n) = \Theta(f(n))$, show that $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

Algorithm $\mathcal{A}$ has an average case complexity $T(n)$

- To prove that $T(n) = O(f(n))$ for a given distribution, show this by **averaging over all** size n inputs for **all** $n \geq n_0$ for some integer n_0.
- To prove that $T(n) = \Omega(f(n))$ for a given distribution, show this by **averaging over all** size n inputs for **infinitely many** n.
- To prove that $T(n) = \Theta(f(n))$ for a given distribution, show that $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

The Prefix-Sum Problem

Input

- An array $A = [A[1], A[2], \ldots, A[n]]$ with $n \geq 1$ real numbers

Output

- An array $S = [S[1], S[2], \ldots, S[n]]$ such that for all $1 \leq i \leq n$,

$$S[i] = \sum_{j=1}^{i} A[j]$$

Example

- $A = [13, 34, \ \ 8, \ \ 3, \ \ 5, 21, \ldots]$
- $S = [13, 47, 55, 58, 63, 84, \ldots]$

Optimization goal

- Minimize the number of **additions** between numbers

A By Definition Algorithm

Algorithm

prefix-sum(A)

```
for i = 1 to n do
    S[i] := 0
for i = 1 to n do
    for j = 1 to i do
        S[i] := S[i] + A[j]
```

Correctness

- By definition

Complexity

- $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ **additions** in the inner loop
- $\frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} = \Theta(n^2)$ complexity

A By Induction Algorithm

Algorithm

prefix-sum(A)

$S[1] := A[1]$

for $i = 2$ to n do

$S[i] :=$ **S[i-1] + A[i]**

Correctness

- Initially, $S[1] = A[1]$
- Induction hypothesis, for $1 \leq i \leq n-1$, after iteration $i-1$:

$$S[i-1] = \sum_{j=1}^{i-1} A[j]$$

- By Induction for $2 \leq i \leq n$, after iteration i:

$$S[i] = S[i-1] + A[i] = \sum_{j=1}^{i-1} A[j] + A[i] = \sum_{j=1}^{i} A[j]$$

A By Induction Algorithm

Algorithm

prefix-sum(A)

$S[1] := A[1]$

for $i = 2$ to n do

$S[i] :=$ **S[i-1] + A[i]**

Complexity

- $n - 1$ additions in the loop
- $\Theta(n)$ complexity

Evaluating a Polynomial

Input

- Real numbers c and $a_0, a_1, \ldots, a_n$ such that $a_n \neq 0$

Output

- The value of the polynomial $P(x)$ for $x = c$:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Example

- $a_4 = 5$, $a_3 = 0$, $a_2 = -7$, $a_1 = 3$, $a_0 = -11$, and $c = 2$
- $P(x) = 5x^4 - 7x^2 + 3x - 11$
- $P(2) = 5 \cdot 2^4 - 7 \cdot 2^2 + 3 \cdot 2 - 11 = 47$

Optimization goal

- Minimize the number of **operations** between real numbers
  - **multiplications** and **additions** and **subtractions**

A By Definition Algorithm

Algorithm

Polynomial-Evaluation$(P(x), c)$

$P(c) = a_0$
for $i = 1$ to n do
  $a = a_i$
  for $j = 1$ to i do
    $a = a \times c$ (* $a = a_i c^i$ *)
  $P(c) = P(c) + a$ (* $P(c) = a_i c^i + a_{i-1} c^{i-1} + \cdots + a_1 c + a_0$ *)
return$(P(c))$ (* $P(c) = a_n c^n + a_{n-1} c^{n-1} + \cdots + a_1 c + a_0$ *)

Correctness

- By definition

A By Definition Algorithm

Complexity

- i multiplications in the i^{th} iteration of the inner loop
- $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ multiplications overall
- n additions in the outer loop
- Total number of operations:

$$\frac{n(n+1)}{2} + n = \frac{n^2+n}{2} + \frac{2n}{2} = \frac{n^2+3n}{2} = \frac{1}{2}n^2 + \frac{3}{2}n$$

- $\Theta(n^2)$ overall complexity.

A Prefix-Sum Algorithm

Idea

- Compute $c, c^2, c^3, \ldots, c^n$ all the powers of c using the efficient **prefix-sum** method

Algorithm

Polynomial-Evaluation$(P(x), c)$

$P(c) = a_0$
$cc = 1$
for $i = 1$ to n do
  $cc = cc \times c$ (* $cc = c^i$ *)
  $P(c) = P(c) + a_i \times cc$ (* $P(c) = a_i c^i + a_{i-1} c^{i-1} + \cdots + a_1 c + a_0$ *)
return$(P(c))$ (* $P(c) = a_n c^n + a_{n-1} c^{n-1} + \cdots + a_1 c + a_0$ *)

Correctness

- By induction

A Prefix-Sum Algorithm

Complexity

- 2 multiplications in the i^{th} iteration of the loop
- 1 addition in the i^{th} iteration of the loop
- Total of $3n$ operations: $2n$ multiplications and n additions
- $\Theta(n)$ overall complexity

The Horner's Algorithm

Idea

- $P(x) = (\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots)x + a_0$

Example I

- $4x^3 + 3x^2 + 2x + 1 = ((4x + 3)x + 2)x + 1$

Example II

- $5x^4 - 7x^2 + 3x - 11 = (((5x + 0)x - 7)x + 3)x - 11$

The Horner's Algorithm

Algorithm

- **Polynomial-Evaluation**$(P(x), c)$

  $P(c) = a_n$
  for $i = n - 1$ downto 0 do
    $P(c) = P(c) \times c + a_i$
    (* $P(c) = a_n c^{n-i} + a_{n-1} c^{n-i-1} + \cdots + a_{i+1} c + a_i$ *)
  return$(P(c))$
  (* $P(c) = a_n c^n + a_{n-1} c^{n-1} + \cdots + a_1 c + a_0$ *)

Correctness

- By Induction

The Horner's Algorithm

Example II

- **Input:** Evaluate $P(x) = 5x^4 - 7x^2 + 3x - 11$ for $c = 2$
  - In the above polynomial $a_3 = 0$

Running the algorithm

$$
\begin{array}{rcrcrcr}
P_4(x) & = & a_4 & = & & = & 5 \\
P_3(x) & = & P_4(x) \cdot c + a_3 & = & 5 \cdot 2 + 0 & = & 10 \\
P_2(x) & = & P_3(x) \cdot c + a_2 & = & 10 \cdot 2 - 7 & = & 13 \\
P_1(x) & = & P_2(x) \cdot c + a_1 & = & 13 \cdot 2 + 3 & = & 29 \\
P(x) & = & P_1(x) \cdot c + a_0 & = & 29 \cdot 2 - 11 & = & 47
\end{array}
$$

The Horner's Algorithm

Complexity

- 1 multiplication in the i^{th} iteration of the loop
- 1 addition in the i^{th} iteration of the loop
- Total of $2n$ operations: n multiplications and n additions
- $\Theta(n)$ overall complexity

A Dictionary Search Problem

Input

- A sorted array A with n keys: $A[1] \leq A[2] \leq \cdots \leq A[n]$
- A key K

Output

- Does K appear in A? **YES** or **NO**
- If **YES**: The first index i such that $A[i] = K$
- If **NO**: The largest index i such that $A[i] < K$ or $i = 0$ if $K < A[1]$

Method

- **Comparisons** between K and the keys in the array
- $K < A[i]$? $K \leq A[i]$? $K = A[i]$? $K \geq A[i]$? $K > A[i]$?

Complexity

- Number of **comparisons**

A Search Game

Game

- **Player 1:** Selects an integer x in the range $[1..n]$
- **Player 2:** Searches for x only with **comparisons** of the type $x \leq i$ for some $1 \leq i \leq n$

Players Goal

- **Player 1** tries to **maximize** the number of **comparisons** until Player 2 finds the value of x
- **Player 2** tries to **minimize** the number of **comparisons** until finding the value of x

Complexity: number of comparisons

- In the worst case or in the average case
- As a function of n

The Two Models are "Equivalent"

Equivalence

- $x \leq i$ is **"equivalent"** to $K \leq A[i]$
- Algorithms can be **"converted"** from one model to another while preserving the complexity

Convenience

- It is **"easier"** to design algorithms in the search game model
- It is **"easier"** to prove bounds and limitations on algorithms in the search game model

Sequential Search

Algorithm outline

- Assume a search for x in the **range** $[1..n]$
- Throughout the algorithm, maintain a lower bound ℓ on x such that $\ell \leq x \leq n$
- Initially, $\ell = 1$
- In each round, compare x with the lower bound ℓ
- If $x > \ell$ then increment ℓ by 1
- If $x \leq \ell$ then return ℓ

Sequential Search

Example

- **Input:** $n = 10$ and $x = 7 \quad \Rightarrow \quad (*\ x \in [1..10]\ *)$
- **Search procedure:**
  - Q1: $x \leq 1 \quad \Rightarrow$ A1: NO $\quad (*\ x \in [2..10]\ *)$
  - Q2: $x \leq 2 \quad \Rightarrow$ A2: NO $\quad (*\ x \in [3..10]\ *)$
  - Q3: $x \leq 3 \quad \Rightarrow$ A3: NO $\quad (*\ x \in [4..10]\ *)$
  - Q4: $x \leq 4 \quad \Rightarrow$ A4: NO $\quad (*\ x \in [5..10]\ *)$
  - Q5: $x \leq 5 \quad \Rightarrow$ A5: NO $\quad (*\ x \in [6..10]\ *)$
  - Q6: $x \leq 6 \quad \Rightarrow$ A6: NO $\quad (*\ x \in [7..10]\ *)$
  - Q7: $x \leq 7 \quad \Rightarrow$ A7: YES $\quad (*\ x \in [7..7]\ \ *)$
- **Output:** $x = 7$
- **Complexity:** 7 comparisons

Sequential Search

Algorithm pseudocode I

Sequential-Search (n, x)

```
ℓ = 1
repeat
    if x ≤ ℓ     (* comparison *)
    then return ℓ
    else ℓ = ℓ + 1
```

Algorithm pseudocode II

Sequential-Search (n, x)

```
ℓ = 1
while x > ℓ do    (* comparison *)
    ℓ = ℓ + 1
return ℓ
```

Sequential Search

Correctness

- By induction, $\ell \leq x \leq n$ after $\ell - 1$ comparisons with a **NO** answer

Termination

- If $x \leq \ell$ then necessarily $x = \ell$ because by the induction hypothesis $x \geq \ell$
- Eventually $x \leq n$

Sequential Search

Worst case complexity

- n **comparisons** in the worst case when $x = n$
- In fact, only $n - 1$ **comparisons** since there is no need for the last comparison when $x = n$

Best case complexity

- Only 1 **comparison** when $x = 1$

Average case complexity

- $(n+1)/2$ **comparisons** on average for a random x selected with a uniform distribution from the range $[1..n]$:

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

Sequential Search

Searching in an array pseudocode

Sequential-Search $(A = [A[1] \leq A[2] \leq \cdots \leq A[n]], K)$

```
if K < A[1] then return (K < A[1])        (* comparison *)
if K > A[n] then return (K > A[n])        (* comparison *)
ℓ = 1
while K > A[ℓ] do                         (* comparison *)
    ℓ = ℓ + 1
if K < A[ℓ]                               (* comparison *)
then return (A[ℓ − 1] < K < A[ℓ])
else return (K = A[ℓ])
```

Worst case number of comparisons

- $n + 3$ comparisons when $K = A[n]$

Binary Search

Algorithm outline

- Assume a search for x in the **range** $[1..n]$
- Throughout the algorithm, maintain a range $[\ell..u]$ such that $\ell \leq x \leq u$
- Initially, $\ell = 1$ and $u = n$
- In each round, compare x with the middle of the range $m = \left\lfloor \frac{u+\ell}{2} \right\rfloor$
- If $x \leq m$ then update $u = m$
- If $x > m$ then update $\ell = m + 1$
- Terminate when $\ell = u$
- Return $x = \ell = u$

Binary Search – Example

- **Input:** $n = 128$ and $x = 50 \quad \Rightarrow \quad (*\ x \in [1..128]\ *)$
- **Search procedure:**
  - Q1: $x \leq 64 \quad \Rightarrow$ A1: YES $(*\ x \in [1..64]\ *)$
  - Q2: $x \leq 32 \quad \Rightarrow$ A2: NO $(*\ x \in [33..64]\ *)$
  - Q3: $x \leq 48 \quad \Rightarrow$ A3: NO $(*\ x \in [49..64]\ *)$
  - Q4: $x \leq 56 \quad \Rightarrow$ A4: YES $(*\ x \in [49..56]\ *)$
  - Q5: $x \leq 52 \quad \Rightarrow$ A5: YES $(*\ x \in [49..52]\ *)$
  - Q6: $x \leq 50 \quad \Rightarrow$ A6: YES $(*\ x \in [49..50]\ *)$
  - Q7: $x \leq 49 \quad \Rightarrow$ A7: NO $(*\ x \in [50..50]\ *)$
- **Output:** $x = 50$
- **Complexity:** $7 = \log_2(128)$ comparisons

Binary Search

Algorithm pseudocode

Binary-Search (n, x)

$\ell = 1$
$u = n$
while $\ell < u$
  $m = \lfloor \frac{u+\ell}{2} \rfloor$
  if $x \leq m$ (* comparison *)
  then $u = m$
  else $\ell = m + 1$
return ℓ

Binary Search

Notations

- Let u_j and ℓ_j be the values of u and ℓ after iteration j of the algorithm
- Let $\Delta_j = u_j - \ell_j + 1$ be the size of the range $[\ell_j..u_j]$
- Initially $\ell_0 = 1$, $u_0 = n$, and $\Delta_0 = n$

Observation

- $\Delta_{j+1} \leq \left\lceil \frac{\Delta_j}{2} \right\rceil$ for $j \geq 0$

Corollary

- $\Delta_k = 1$ for $k = \lceil \log_2 n \rceil$

Binary Search – Correctness and Complexity

Correctness

- By induction, always $\ell \leq x \leq u$
- At the end, $\Delta = 1$ and therefore $\ell = u$ which implies that $x = \ell = u$

Complexity

- There are at most $\lceil \log_2 n \rceil$ iterations and one **comparison** per iteration
- Therefore, the **worst-case** complexity is $\lceil \log_2 n \rceil$
- If n is not a power of 2, then for some x there are only $\lfloor \log_2 n \rfloor$ iterations
- Therefore, the **average-case** complexity is approximately $\log_2 n$

Binary Search

Searching in an array pseudocode

Binary-Search $(A = [A[1] \leq A[2] \leq \cdots \leq A[n]], K)$

```
if K < A[1] then return (K < A[1])        (* comparison *)
if K > A[n] then return (K > A[n])        (* comparison *)
ℓ = 1 and u = n
while ℓ < u
    m = ⌊(u+ℓ)/2⌋
    if K ≤ A[m]                           (* comparison *)
    then u = m
    else ℓ = m + 1
if K < A[ℓ]                               (* comparison *)
then return (A[ℓ − 1] < K < A[ℓ])
else return (K = A[ℓ])
```

Number of comparisons

- $\lceil \log_2 n \rceil + 3$ comparisons

Binary-Search vs. Sequential-Search

	Binary-Search	Sequential-Search
Best-Case	$\lfloor \log_2 n \rfloor$	1
Worst-Case	$\lceil \log_2 n \rceil$	$n - 1$
Average-Case	$\approx \log_2 n$	$\approx n/2$

Adversary Player I

Goal

- Maximize the number of **comparisons** until **Player 2** finds x

Strategy

- **Player 1 does not** select x at the beginning of the game; instead, it maintains a set $\mathcal{S}$ of candidates for x
- Given a search question:
  - $\mathcal{S}(Y)$ – the set of candidates if the answer is **YES**
  - $\mathcal{S}(N)$ – the set of candidates if the answer is **NO**
- The adversary answer rule:
  - **YES** if $|\mathcal{S}(Y)| \geq |\mathcal{S}(N)|$
  - **NO** if $|\mathcal{S}(Y)| < |\mathcal{S}(N)|$

Adversary Player I

Example: A possible algorithm

- **Input:** $n = 34$ (* $x \in [1..34]$ *)
- **Search:**
  - Q1: $x \leq 13 \Rightarrow$ A1: NO (* $x \in [14..34]$ *)
  - Q2: $x \leq 26 \Rightarrow$ A2: YES (* $x \in [14..26]$ *)
  - Q3: $x \leq 18 \Rightarrow$ A3: NO (* $x \in [19..26]$ *)
  - Q4: $x \leq 23 \Rightarrow$ A4: YES (* $x \in [19..23]$ *)
  - Q5: $x \leq 20 \Rightarrow$ A5: NO (* $x \in [21..23]$ *)
  - Q6: $x \leq 22 \Rightarrow$ A6: YES (* $x \in [21..22]$ *)
  - Q7: $x \leq 21 \Rightarrow$ A7: YES (* $x \in [21..21]$ *)
- **Output:** $x = 21$

Adversary Player I

Example: Binary-Search

- **Input:** $n = 34$ (* $x \in [1..34]$ *)
- **Search:**
  - Q1: $x \leq 17 \Rightarrow$ A1: YES (* $x \in [1..17]$ *)
  - Q2: $x \leq 9 \Rightarrow$ A2: YES (* $x \in [1..9]$ *)
  - Q3: $x \leq 5 \Rightarrow$ A3: YES (* $x \in [1..5]$ *)
  - Q4: $x \leq 3 \Rightarrow$ A4: YES (* $x \in [1..3]$ *)
  - Q5: $x \leq 2 \Rightarrow$ A5: YES (* $x \in [1..2]$ *)
  - Q6: $x \leq 1 \Rightarrow$ A6: YES (* $x \in [1..1]$ *)
- **Output:** $x = 1$

Observation

- With Binary-Search the search always ends up with $x = 1$

Impossible to Search Faster than Binary Search

Theorem

- There exists $1 \leq x \leq n$ for which the **Adversary Player 1** forces **Player 2** to ask at least $\lceil \log_2 n \rceil$ comparisons

Proof

- Assume that **Player 2** asks k comparisons to find x
- Let $\mathcal{S}_i$ be the set of candidates after i comparisons
- In particular, $|\mathcal{S}_0| = n$ and $|\mathcal{S}_k| = 1$
- $\mathcal{S} = \mathcal{S}(Y) \cup \mathcal{S}(N)$ implies that $|\mathcal{S}_{i+1}|/|\mathcal{S}_i| \geq (1/2)$ for $1 \leq i \leq k-1$
- $\lceil \log_2 n \rceil$ rounds are required to decrease n to 1 by halving
- Therefore, $k \geq \lceil \log_2 n \rceil$

Remarks

Worst case

- The $\lceil \log_2 n \rceil$ lower bound is a **worst case** bound
- No algorithm can guarantee less comparisons **for all** values of x

Average case

- It is possible to prove an $\Omega(\log n)$ **average case** lower bound

Other search models

- The theorem holds for a **"stronger" Player 2** – one that may ask any **YES/NO** questions. For example,
  - Is x even?
  - Is x a prime number?
  - Does $x \in \{1, 2, 3, 5, 8, 13, 21, 34\}$?

Searching with "Clues"

Clue

- **Player 1** selects only even numbers $2, 4, 6, 8, \ldots$ between 1 and an even n

A modified Binary Search

- The search domain is $1, 2, \ldots, n/2$
- Instead of asking **"if** $x \leq i$**"**, **Player 2** asks **"if** $x \leq 2i$**"** and then considers the answer as if it was the answer to **"if** $x \leq i$**"**
- When the search outputs $x = i$ the modified search outputs $2i$

Complexity

- $\lceil \log_2(n/2) \rceil \approx \log_2(n/2) = \log_2(n) - 1$ comparisons
- The saving is only 1 comparison although the clue "eliminated" about half of the candidates!

Searching with "Clues"

Clue

- **Player 1** selects only even numbers $2, 4, 6, 8, \ldots$ between 1 and and an even n

Example

- $n = 32 = 2 \cdot 16$ and $x = 20 = 2 \cdot 10$
- The possible 16 values for x are $2, 4, 6, \ldots, 32$ and the search domain is $1, 2, \ldots, 16$

Running the algorithm

- Question 1: $x \leq (2 \cdot 8 = 16)$? because $8 = \lfloor (1 + 16)/2 \rfloor$
- Question 2: $x \leq (2 \cdot 12 = 24)$? because $12 = \lfloor (9 + 16)/2 \rfloor$
- Question 3: $x \leq (2 \cdot 10 = 20)$? because $10 = \lfloor (9 + 12)/2 \rfloor$
- Question 4: $x \leq (2 \cdot 9 = 18)$**?** because $9 = \lfloor (9 + 10)/2 \rfloor$
- $x = 20$ found with $4 = \log_2 16 = \log_2 32 - 1$ comparisons

Searching with "Clues"

Clue

- **Player 1** selects only square numbers $1, 4, 9, 16, \ldots$ between 1 and a square number n

A modified Binary Search

- The search domain is $1, 2, \ldots, \sqrt{n}$
- Instead of asking **"if $x \leq i$"**, **Player 2** asks **"if $x \leq i^2$"** and then considers the answer as if it was the answer to **"if $x \leq i$"**
- When the search outputs $x = i$ the modified search outputs i^2

Complexity

- $\lceil \log_2(\sqrt{n}) \rceil \approx \log_2(\sqrt{n}) = \frac{1}{2} \log_2(n)$ comparisons
- The saving is only half of the comparisons although the clue "eliminated" almost all the candidates!

Searching with "Clues"

Clue

- **Player 1** selects only square numbers $1, 4, 9, 16, \ldots$ between 1 and a square number n

Example

- $n = 256 = 16^2$ and $x = 100 = 10^2$
- The possible 16 values for x are $1, 4, 9, \ldots, 256$ and the search domain is $1, 2, \ldots, 16$

Running the algorithm

- Question 1: $x \leq (8^2 = 64)?$ because $8 = \lfloor (1 + 16)/2 \rfloor$
- Question 2: $x \leq (12^2 = 144)?$ because $12 = \lfloor (9 + 16)/2 \rfloor$
- Question 3: $x \leq (10^2 = 100?)$ because $10 = \lfloor (9 + 12)/10 \rfloor$
- Question 4: $x \leq (9^2 = 81?)$ because $9 = \lfloor (9 + 10)/10 \rfloor$
- $x = 100$ found with $4 = \log_2 16 = (1/2) \log_2 256$ comparisons

Searching with "Clues"

Clue

- **Player 1** selects only powers of 2 numbers $2, 4, 8, 16, \ldots$ between 2 and a power of 2 number n

A modified Binary Search

- The search domain is $1, 2, \ldots, \log_2 n$
- Instead of asking **"if** $x \leq i$**"**, **Player 2** asks **"if** $x \leq 2^i$**"** and then considers the answer as if it was the answer to **"if** $x \leq i$**"**
- When the search outputs $x = i$ the modified search outputs 2^i

Complexity

- $\lceil \log_2(\log_2(n)) \rceil \approx \log_2(\log_2(n))$ comparisons
- For $n = 2^{32} = 4294967296$ the saving is from 32 to 5 comparisons although there are only 32 candidates!

Searching with "Clues"

Clue

- **Player 1** selects only powers of 2 numbers $2, 4, 8, 16, \ldots$ between 2 and a power of 2 number n

Example

- $n = 65536 = 2^{16}$ and $x = 1024 = 2^{10}$
- The possible 16 values for x are $2, 4, 8, \ldots, 65536$ and the search domain is $1, 2, \ldots, 16$

Running the algorithm

- Question 1: $x \leq (2^8 = 256)?$ $8 = \lfloor (1 + 16)/2 \rfloor$
- Question 2: $x \leq (2^{12} = 4096)?$ $12 = \lfloor (9 + 16)/2 \rfloor$
- Question 3: $x \leq (2^{10} = 1024?)$ $10 = \lfloor (9 + 12)/10 \rfloor$
- Question 4: $x \leq (2^9 = 512?)$ $9 = \lfloor (9 + 10)/10 \rfloor$
- $x = 1024$ found with $4 = \log_2 16 = \log_2 \log_2 65536$ comparisons

Searching an Unbounded Domain

Game

- **Player 1:** Selects any positive integer x
- **Player 2:** Searches for x with **comparisons** $x \leq i$ for some integer i

Adversary Player 1

- Always answers **NO**
- **Player 2** will never find x!

Player 2 Goal

- Find x with as minimum possible comparisons as a function of x
- Ask "**less**" comparisons when x is small and ask "**more**" comparisons when x is large

Searching an Unbounded Domain

Sequential search

- Sequential search finds x with exactly $\underline{\mathbf{x}}$ comparisons

The doubling technique

- A strategy that finds x with approximately $\underline{\mathbf{2}\log_{\mathbf{2}}(\mathbf{x})}$ comparisons

A more sophisticated doubling technique

- A strategy that finds x with approximately $\underline{\log_{\mathbf{2}}(\mathbf{x}) + \mathbf{2}\log_{\mathbf{2}}\log_{\mathbf{2}}(\mathbf{x})}$ comparisons

Optimal solution

- A strategy that finds x with approximately

$$\underline{\log_{\mathbf{2}}(\mathbf{x}) + \log_{\mathbf{2}}\log_{\mathbf{2}}(\mathbf{x}) + \log_{\mathbf{2}}\log_{\mathbf{2}}\log_{\mathbf{2}}(\mathbf{x}) + \cdots}$$

comparisons

The Doubling Technique

Strategy

- **Phase 1:** Ask the following comparisons until the answer is **YES**:

$$x \leq 1?\ \ x \leq 2?\ \ x \leq 4?\ \ x \leq 8?\ \ \cdots\ \ x \leq 2^j?\ \ \cdots$$

- Assume $2^{k-1} < x \leq 2^k$
- **Phase 2:** Apply binary search on the domain

$$[2^{k-1} + 1..2^k]$$

Complexity

- $k + 1$ comparisons are asked in **Phase 1**
- The number of comparisons asked in **Phase 2** is

$$\left\lceil \log_2(2^k - (2^{k-1} + 1) + 1) \right\rceil = \left\lceil \log_2(2^{k-1}) \right\rceil = k - 1$$

- Total number of comparisons:

$$(k + 1) + (k - 1) = 2k = 2\lceil \log_2(x) \rceil$$

The Doubling Technique – Example

- **Input:** $x = 50$
- **Search procedure:**
  - Q1: $x \leq 1$ $\Rightarrow$ A1: NO (* $x \in [2..\infty]$ *).
  - Q2: $x \leq 2$ $\Rightarrow$ A2: NO (* $x \in [3..\infty]$ *).
  - Q3: $x \leq 4$ $\Rightarrow$ A3: NO (* $x \in [5..\infty]$ *).
  - Q4: $x \leq 8$ $\Rightarrow$ A4: NO (* $x \in [9..\infty]$ *).
  - Q5: $x \leq 16$ $\Rightarrow$ A5: NO (* $x \in [17..\infty]$ *).
  - Q6: $x \leq 32$ $\Rightarrow$ A6: NO (* $x \in [33..\infty]$ *).
  - Q7: $x \leq 64$ $\Rightarrow$ A7: YES (* $x \in [33..64]$ *).
  - Q8: $x \leq 48$ $\Rightarrow$ A8: NO (* $x \in [49..64]$ *).
  - Q9: $x \leq 56$ $\Rightarrow$ A9: YES (* $x \in [49..56]$ *).
  - Q10: $x \leq 52$ $\Rightarrow$ A10: YES (* $x \in [49..52]$ *).
  - Q11: $x \leq 50$ $\Rightarrow$ A11: YES (* $x \in [49..50]$ *).
  - Q12: $x \leq 49$ $\Rightarrow$ A12: NO (* $x \in [50..50]$ *).
- **Output:** $x = 50$
- **Complexity:** $12 = \lceil 2 \log_2 50 \rceil$ comparisons