Coverage of CSE 4403: Algorithms

Asaduzzaman Herok

June 28, 2025

Contents

1 May 06, 2025: Introduction

1.1 What is an Algorithm

In computer science, A method that a computer can use for the solution of a problem.
Definition: An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. Any well-defined computational procedure that takes some value or set of values as input and produces some value or set of values as output, in a finite amount of time.

All algorithms must satisfy the following criteria: [P:-01 of [1]]

1. Input
2. Output
3. Definiteness
4. Finiteness in time
5. Effectiveness

The four distinct areas of study: [P:-03 of [1]]

1. How to devise an algorithm
2. how to validate an algorithm
3. How to analyse an algorithm
4. How to test an algorithm

1.2 Algorithm Specification

Pseudocode Conventions: [P:-05 of [1]]

1. Comment
2. Block
3. Identifier, Compound data as records.
4. Assignment
5. Logical Comparison
6. Multidimensional Arrays
7. Loops: For, while, do while
8. Conditionals if then else
9. Case
10. Input/output read write
11. Algorithm Procedure

Example problem: Given a set of numbers `A` and a number `x`, find if the number exists in set `A` or not.

```
Algorithm Find(A,x)
{
    length:= A.length;
    For i:=0 to length step 1 do
    {
        if A[i]=x then
            return True;
    }
    return False;
}
```

2 May 07, 2025: Recursive Algorithms

2.1 Recursive Algorithms

A recursive function is a function that is defined in terms of itself. Similarly an algorithm is said to be recursive if the same algorithm is invoked in the body. Two types of recursions:

1. Direct recursive: The algorithm that calls itself.
2. Indirect recursive: The algorithm **A** calls another algorithm **B** and from be **A** is called again.

Every looping problem can be solved using Recursion.
How to approach recursive problems? Examples:

- Calculating Binomial Coefficients: ${}^{n}C_r = {}^{n-1}C_r + {}^{(n-1)}C_{r-1}$
- Calculating GCD: $GCD(x, y) = GCD(y, x \bmod y)$
- Solving Tower of Hanoi problem. [P:-11 of [1]]
- Generating all possible combination. [P:-13 of [1]]

Practice recursive problems [3]

3 May 13, 2025: Performance Analysis & Measurement

Recap of previous lecture & completing Generate all possible combination problem.

3.1 Performance analysis

Criteria for judging an algorithm:

- Does it do what we want it to do?
- Does it work correctly according to the original specification of the task?
- Is there documentation that describes how to use it and how it works?
- Are procedures created in such a way that they perform logical sub-funcitons?
- Is the code readable?

These are vitally important when it comes to writing software for large systems. Since we are working with smaller classical algorithms we are not considering them. Instead there are other criteria for judging algorithms that have a more direct relationship to performance.

- Space Complexity: It is the amount of memory needed to run the algorithm from starting to completion.
- Time Complexity: It is the amount of time needed to runt the algorith from starting to completion.

The performance evaluation has two phase: (1) a Priori estimates (performance analysis) and (2) a posteriori testing (performance evaluation).

3.1.1 Space Complesity:

```
Algorithm abc(a,b,c)
{
    return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

```
Algorithm Sum(a,n)
{
    s:=0.0;
    for i:=1 to n do
        s:=s+a[i];
    return s;
}
```

```
Algorithm RSum(a, n)
{
    if(n<=0) then return 0.0;
    else return RSum(a,n-1)+a[n];
}
```

The space needed for each of the algorithm is the sum of two major components:

- A fixed part which is independent of characteristics (number, size) of the inputs and outputs. The instruction space, simple variables, fixed-size components variables, space for constants and so on.
- A variable part which is the space needed by component variables whose size is dependent on the particular problem instance being solved. the space needed by referenced variable, recursion stack space.

The space requirement $S(P)$ of any algorithm P is

$$S(P) = c + S_p(instance\ charactersistics)$$

where c is a constant. While analyzing we concentrate on $S_p(instance\ charactersistics)$. Examples [P:-17 of [1]]

3.1.2 Time complexity:

The time $T(P)$ of algorithm P is consist of compile time and run time. The compile time doesn't depends on the problem instance. So we concentrate on the run time only denoted by $t_p(instance\ characteristics)$.

$$t_p(n) = c_a ADD(n) + c_s(SUB(n) + c_m MUl(n) + c_d(DIV)(n) + ...$$

Obtaining such exact formula is itself a impossible task. So we count only the number of program steps.
Program Step: is loosely defined as syntactically for semenatically meaningful segment of a program that has an independent execution time. Example:

```
return a+b+b*c+(a+b-c)/(a+b)+4.0;
```

The number of steps any program depends on the kind of statements. Comments counted as zero steps, assigment without any calls to function is counted as one steps. For iterative loops we consider the step counts only for the control part of the statments.

```
-------------------------------------------------------
Statement               | s/e | Frequency | total steps |
Algorithm Sum(a,n)      |  0  | --        |     0       |
{                       |  0  | --        |     0       |
    s:=0.0;             |  1  | 1         |     1       |
    for i:=1 to n do    |  1  | n+1       |     n+1     |
        s:=s+a[i];      |  1  | n         |     n       |
    return s;           |  1  | 1         |     1       |
}                       |  0  | --        |     0       |
-------------------------------------------------------
Total                                       2n+3
```

Statement	s/e	frequency		total steps	
		$n=0$	$n>0$	$n=0$	$n>0$
1 **Algorithm** RSum(a,n)	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a,n-1)+a[n]$;	$1+x$	0	1	0	$1+x$
7 }	0	–	–	0	0
Total				2	$2+x$

$$x = t_{\mathsf{RSum}}(n-1)$$

Figure 1: Step count of RSum

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a,b,c,m,n)	0	–	0
2 {	0	–	0
3 **for** $i := 1$ **to** m **do**	1	$m+1$	$m+1$
4 **for** $j := 1$ **to** n **do**	1	$m(n+1)$	$mn+m$
5 $c[i,j] := a[i,j] + b[i,j]$;	1	mn	mn
6 }	0	–	0
Total			$2mn+2m+1$

Figure 2: Step count of Matrix Addition

3.2 Asymptotic Notation (O, Ω, Θ)

Big "oh" (O): The function $f(n) = O(g(n))$ if and only if there exist a positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all $n, n \geq n_0$. [P:-29 of [1]]
Big "Omega" (Ω): The function $f(n) = \Omega(g(n))$ if and only if there exist a positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all $n, n \geq n_0$. [P:-30 of [1]]

Big "Theta" (Θ): The function $f(n) = \Theta(g(n))$ if and only if there exist a positive constants c_1, c_2 and n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n, n \geq n_0$. [P:-56 of [2]]

4 May 14, 2025: Divide and Conquer

Given a function to compute on n inputs the *divide-and conquer* strategy splits the inputs into k distinct subsets, $1 < k \leq n$, yielding k subproblems. These sub problems are solved and the sub solutions are combined to generate the whole solution.
Often sub-problems are of the same type as the original problem, so for these cases the reapplication of *divide-and-conquer* principle can be expressed as recursive algorithm.

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
    divide P into smaller instances P1, P2, .. .. Pk, k >1;
    Apply DAndC to each of these subproblems.

    return Combine(DAndC(P1), DAndC(P2), DAndC(P3), ... ... DAndC(Pk))
    }
}
```

$$T(n) = \begin{cases} g(n) & n \; small \\ T(n_1) + T(n_2) + ...T(n_k) + f(n) & otherwise \end{cases}$$

4.1 Binary Search

```
Algorithm BinarySearch(P, l, r)
{
    if Small(P, l, r) then return S(P, l, r);
    else
    {
        m:= (l+r)/2;
        if f(P,l,m) then l:=m;
        else r:= m-1;
    }
}
```

Complexity $O(\log_2 n + f(n))$
For more see [4] and [P:-131 of [1]]

4.2 MergeSort

See the pseudo-code and complexity analysis from [P:-145-149 of [1]]

4.3 Finding Maximum and Minimum

See the pseudo-code and complexity analysis from [P:-139-144 of [1]]

5 May 20, 2025: Divide and Conquer

5.1 Peak Finding 1D and 2D

The followings lecture notes are taken from [5].

Peak Finder

One-dimensional Version

Position 2 is a peak if and only if $b \geq a$ and $b \geq c$. Position 9 is a peak if $i \geq h$.

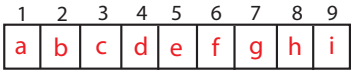
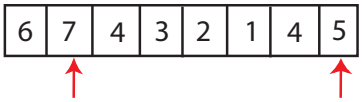


Figure 1: a-i are numbers

Problem: Find a peak if it exists (Does it always exist?)

Straightforward Algorithm



Start from left

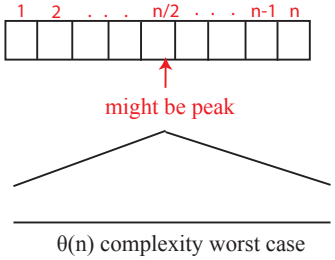
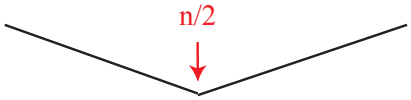


Figure 2: Look at $n/2$ elements on average, could look at n elements in the worst case

What if we start in the middle? For the configuration below, we would look at $n/2$ elements. Would we have to ever look at more than $n/2$ elements if we start in the middle, and choose a direction based on which neighboring element is larger that the middle element?



Can we do better?

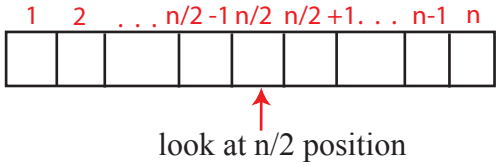


Figure 3: Divide & Conquer

- If $a[n/2] < a[n/2-1]$ then only look at left half $1 \ldots n/2 - - - 1$ to look for peak
- Else if $a[n/2] < a[n/2+1]$ then only look at right half $n/2+1 \ldots n$ to look for peak
- Else $n/2$ position is a peak: WHY?

$$\begin{aligned} a[n/2] &\geq a[n/2-1] \\ a[n/2] &\geq a[n/2+1] \end{aligned}$$

What is the complexity?

$$T(n) = T(n/2) + \underbrace{\Theta(1)}_{\text{to compare a[n/2] to neighbors}} = \Theta(1) + \ldots + \Theta(1) \ (\log_2(n) \ times) = \Theta(\log_2(n))$$

In order to sum up the $\Theta(i)$'s as we do here, we need to find a constant that works for all. If $n = 1000000, \Theta(n)$ algo needs 13 sec in python. If algo is $\Theta(\log n)$ we only need 0.001 sec. Argue that the algorithm is correct.

Two-dimensional Version

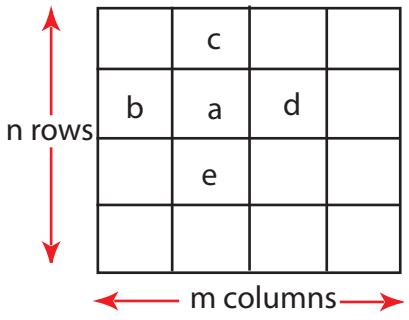


Figure 4: Greedy Ascent Algorithm: $\Theta(nm)$ complexity, $\Theta(n^2)$ algorithm if $m = n$

a is a 2D-peak iff $a \geq b, a \geq d, a \geq c, a \geq e$

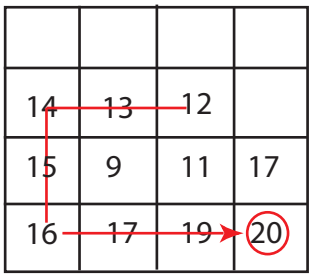
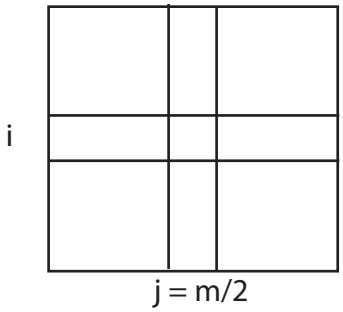


Figure 5: Circled value is peak.

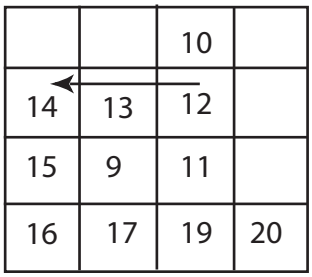
Attempt # 1: Extend 1D Divide and Conquer to 2D



- Pick middle column $j = m/2$.
- Find a 1D-peak at i, j.
- Use (i, j) as a start point on row i to find 1D-peak on row i.

Attempt #1 fails

Problem: 2D-peak may not exist on row i

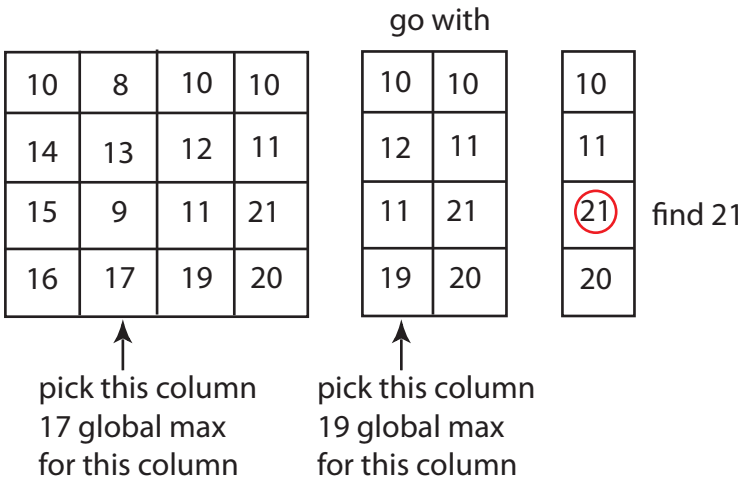


End up with 14 which is not a 2D-peak.

Attempt # 2

- Pick middle column $j = m/2$
- Find global maximum on column j at (i, j)
- Compare $(i, j-1), (i, j), (i, j+1)$
- Pick left columns of $(i, j-1) > (i, j)$
- Similarly for right
- (i, j) is a 2D-peak if neither condition holds ← WHY?
- Solve the new problem with half the number of columns.
- When you have a single column, find global maximum and you're done.

Example of Attempt #2



Complexity of Attempt #2

If $T(n, m)$ denotes work required to solve problem with n rows and m columns

$$
\begin{aligned}
T(n,m) &= T(n, m/2) + \Theta(n) \text{ (to find global maximum on a column — (n rows))} \\
T(n,m) &= \underbrace{\Theta(n) + \ldots + \Theta(n)}_{\log m} \\
&= \Theta(n \log m) = \Theta(n \log n) \text{ if m = n}
\end{aligned}
$$

Question: What if we replaced global maximum with 1D-peak in Attempt #2? Would that work?

6 May 27, 2025: Complexity analysis of Divide and Conquer

6.1 Solving Recurrence [P:-80 of [2]]

Divide and Conquer algorithms typically divide a problem into smaller subproblems, solve each recursively, and combine their solutions. This often leads to recurrence relations of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:

- a = number of subproblems
- n/b = size of each subproblem
- $f(n)$ = cost of dividing and combining the subproblems

We study three techniques to analyze such recurrences:

6.1.1 Substitution Method [P:-90-94 of [2]]

Idea: Guess the form of the solution and prove it by induction.
Example: Given

$$T(n) = 2T(n/2) + n$$

Guess: $T(n) = O(n \log n)$
Proof:
Assume the inductive hypothesis that $T(n) \leq cn \log n$ for all $n \geq n_0$, where $c > 0$ and $n_0 > 0$
Then,

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq 2c(n/2)\log(n/2) + n \\ &= cn\log(n/2) + n \\ &= cn(\log n - 1) + n \\ &= cn\log n - cn + n \\ &\leq cn\log n \qquad \text{for large } c \end{aligned}$$

Thus, $T(n) = O(n \log n)$.

6.1.2 Recurrence Tree [P:-95-101 of [2]]

Idea: Build a tree where each node represents the cost at a level of recursion and sum all levels.
Example: For $T(n) = 2T(n/2) + n$,

- Level 0: Work = n
- Level 1: Work = $2 \cdot (n/2) = n$
- Level 2: Work = $4 \cdot (n/4) = n$
- ...(repeats for $\log n$ levels)

Total work = $n \cdot \log n \Rightarrow T(n) = \Theta(n \log n)$

6.1.3 Master Method [P:-101-106 of [2]]

The Master Theorem solves recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Compare $f(n)$ with $n^{\log_b a}$:

Case 1: If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$,
then $T(n) = \Theta(n^{\log_b a})$

Case 2: If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$,
then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Case 3: If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and $af(n/b) \leq cf(n)$ for some $c < 1$ and sufficiently large n,
then $T(n) = \Theta(f(n))$

Example:

$$T(n) = 2T(n/2) + n \qquad \text{where} \quad a = 2, b = 2, f(n) = n$$

$$n^{\log_2 2} = n \Rightarrow f(n) = \Theta(n) \quad (\text{Case 2}) \Rightarrow T(n) = \Theta(n \log n)$$

6.2 Greedy

The **Greedy Method** is a problem-solving technique that builds a solution piece by piece, choosing the *locally optimal choice* at each step with the hope of finding a global optimum. It is efficient and intuitive but works correctly only for problems exhibiting specific properties.

6.3 Key Properties

- **Greedy-choice property**: A globally optimal solution can be arrived at by making a sequence of locally optimal choices.
- **Optimal substructure**: The optimal solution contains optimal solutions to its subproblems.

7 May 28, 2025: General Greedy Algorithm Framework

Most, though not all problems have n input and require us to to obtain a subset that satisfies some constraints.
Feasible solution: A feasible solution is any solution that satisfies all the constraints of the problem. It may not be the best solution, but it is valid.
Objective Function: The objective function defines what we want to maximize or minimize. It maps a feasible solution to a numerical value. I is used to evaluate how "good" a solution is.
Optimal Solution: An optimal solution is a feasible solution that optimizes the objective function (maximizes or minimizes it, depending on the problem). It is the best possible solution among all feasible ones.
[P:-197 of [1]]

7.1 Subset Paradigm

Generating and evaluating subsets of a given set to find a feasible and/or optimal solution. [P:-197 of [1]]

```
Algorithm Greedy(a,n)
{
    solution:= {}; // Empty set
    for i:=1 to n do
    {
        x:=Select(a);
        if Feasible(solution,x) then
            solution:= Union(solution, x);
    }
    return solution;
}
```

7.2 Ordering Paradigm

For problems that do not call for the selection of an optimal subset, rather we make decisions by considering the inputs in some order. Each decision is made using an optimization criterion that can be computed using decisions already made.

```
Initialize solution set S = ∅
While solution incomplete:
  Choose best candidate according to greedy criteria
  If candidate is feasible, add it to S
Return S
```

7.3 Fractional KnapSack [P:-198-199 of [1]]

From n objects and a capacity m where object i has a weight w_i and a profit value p_i per weight unit. If a fraction $x_i,\ (0 \le x \le 1)$, of object i is placed in the capacity m then a profit of $p_i x_i$ is earned.
Objective function:

$$\text{Maximize} \sum_{1 \le i \le n} p_i x_i$$

Constriant:

$$\sum_{1 \le i \le n} w_i x_i \le m$$

and

$$x_i,\ (0 \le x \le 1), \quad 1 \le i \le n$$

8 June 17, 2025: General Greedy Algorithm Framework

8.1 Job Sequencing with Deadline [P:-208-212 of [1]]

There are n job. Each job J_i has: a deadline d_i a profit p_i (earned only if the job is finished before or on its deadline). Each job takes **exactly 1 unit of time** Maximize total profit, such that only one job is executed at a time, and each job completes before its deadline.

$$n = 4, \quad P = \{100, 10, 15, 27\} \quad d = \{2, 1, 2, 1\}$$

$$\text{maximize} \sum_{i \in J} P_i$$

8.1.1 Theorem 4.3 and Proof [P:-209 of [1]]

Let J be a set of k jobs and $\sigma = i_1, i_2, .., i_k$ a permutation of jobs in J such that $d_{i_1} \leq d_{i_2} \leq d_{i_3} \leq ... \leq d_{i_k}$. J is a feasible solution iff the jobs in J can be processed in that order σ without violating any deadline.

8.1.2 Greedy Strategy

```
Sort all jobs by decreasing profit.
For each job:
  Try to schedule it in the latest available time slot before its deadline.
  If that slot is free, assign it.
  If not, skip it.
```

9 June 24, 2025: Greedy Algorithm Ordering Paradigm

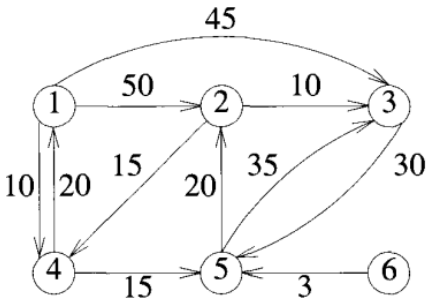
9.1 Optimal Storage on Tabpes [P:-229-232 of [1]]

There are n programs that are to be stored on a computer tape of length l. Each program i has a length l_i, $(1 \leq i \leq n)$. All programs can be stored on the tape if and only if the sum of the lengths of the programs is at most l. Whenever a program is retrieved from the tape, the tape is initially positioned at the front. Hence, if the programs are stored in the order $I = i_1, i_2, ..., i_n$, the time t_j needed to retrieve program i_j is proportional to $\sum_{1 \leq k \leq j} l_{i_k}$. The expected or mean retrieval time (MRT) is $(1/n) \sum_{1 \leq k \leq j} l_{i_k}$. So this is in ordering paradigm. The objective function

$$d(I) = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} l_{i_k}$$

Example: Let $n = 3$ and $(l_1, l_2, l_3) = (5, 10, 3)$.
N.B: Theorems excluded.

9.2 Single Source Shortest Path [P:-241-247 of [1]]



(a) Graph

	Path	*Length*
1)	1, 4	10
2)	1, 4, 5	25
3)	1, 4, 5, 2	45
4)	1, 3	45

(b) Shortest paths from 1

Figure 3: Single Source Shortest Path

Though path generation is not included in the book, it is included in the syllabus. Steps including path track:
Goal: Find the shortest path from a source node to all other nodes in a weighted graph with non-negative edge weights.

Steps

1. **Initialize:**
   - Set distance to source = 0; others = ∞.
   - Set parent of all nodes to `null`.
   - Use a priority queue to process nodes with the smallest distance.
2. **Process Nodes:**
   - For each neighbor of the current node:
     - If a shorter path is found:
       * Update distance.
       * Set parent of the neighbor to the current node.
3. Repeat until all nodes are visited or the destination is reached.
4. **Path Reconstruction:**
   - Backtrack from the destination using the parent array to find the shortest path.

10 June 25, 2025: Dynamic Programming

Dynamic Programming (DP) is a method used in computer science to solve complex problems by breaking them down into simpler subproblems, solving each subproblem just once, and storing the results for future use (usually in a table or memo).

10.1 Characteristic of DP

1. **Overlapping Subproblems**: The same subproblems are solved multiple times. A problem has overlapping subproblems if the solution involves solving the same subproblem multiple times.
2. **Optimal Substructure**: The optimal solution to the problem can be formed by combining optimal solutions of its subproblems. A problem has optimal substructure if an optimal solution to the whole problem can be constructed from optimal solutions of its subproblems.

10.2 Conversion of Recursive Fibonacci to Fibonacci DP [6]

10.3 Optimal Rod cutting [P:-362-372 of [2]]

Books

[1] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms*, en, 2nd ed. New York, NY: W.H. Freeman, 1997.

[2] T. H. Cormen and C. E. Leiserson, *Introduction to Algorithms, fourth edition*, en. London, England: MIT Press, 2022.

Online

[3] *Recursive practice problems with solutions*, en, https://www.geeksforgeeks.org/recursion-practice-problems-solutions/, Accessed: 2025-5-7.

[4] *Binary search - algorithms for competitive programming*, en, https://cp-algorithms.com/num_methods/binary_search.html, Accessed: 2025-5-20.

[5] Lecture Overview, *Lecture 1: Introduction and peak finding*, https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-fall-2011/c32185c7158955425455159a8455298b_MIT6_006F11_lec01.pdf, Accessed: 2025-5-20.

[6] *Nth fibonacci number*, en, https://www.geeksforgeeks.org/dsa/program-for-nth-fibonacci-number/, Accessed: 2025-6-28, Mar. 2011.