

# Algorithms Course Proposed Schedule (Academic Year 2025–2026)

Jair Wuilloud

September 2025

## Содержание

<b>1</b>	<b>First Part of Semester</b>	<b>3</b>
1.1	RAM Model . . . . .	3
1.2	Big O Notations . . . . .	3
1.3	Master Theorem and Derivation . . . . .	3
1.4	Master Theorem, Applications and Restriction . . . . .	4
1.5	Multiplication Algorithms . . . . .	4
1.5.1	Standard multiplication in basis 10 . . . . .	4
1.5.2	Matrix Multiplication . . . . .	5
1.5.3	Paesant Multiplication . . . . .	5
1.6	Computations: $n!$ , Fibonacci, ... . . . .	6
1.7	Knapsack 0–1 . . . . .	6
1.8	Merge Sort . . . . .	6
1.9	Quick Sort . . . . .	7
1.10	Heap Sort . . . . .	7
1.11	Dynamical Computing . . . . .	7
1.12	Greedy Computing . . . . .	8
1.13	Edit (Levenshtein) Distance . . . . .	8
1.14	Explain Divide and Conquer . . . . .	8
<b>2</b>	<b>Second Part of Semester</b>	<b>8</b>
2.1	Course Allocation Problem . . . . .	8
2.2	Huffman Encoding . . . . .	9
2.3	Graph . . . . .	9
2.4	Tree . . . . .	9
2.5	Tree Representation . . . . .	10
2.6	Graph Operations . . . . .	10
2.7	Operations on Graphs . . . . .	10
2.8	Advanced Concepts . . . . .	10
2.9	Transformations of Graphs . . . . .	10
2.10	Algorithm: Topological Sort . . . . .	10
2.11	Algorithm: Cycle Detection . . . . .	10
2.12	Find Strong Components . . . . .	11
2.13	Cut Theorem . . . . .	11

2.14	Find Minimum Spanning Tree . . . . .	11
2.14.1	Cut Property / Theorem . . . . .	11
2.14.2	Kruskal . . . . .	11
2.14.3	Jarnik Algorithm . . . . .	11
2.15	Graph Colouring . . . . .	12
2.15.1	Greedy Algorithm . . . . .	12
2.16	Dijkstra Algorithm . . . . .	12
2.17	Finite Functions . . . . .	13
2.18	Turing . . . . .	13
2.19	P, NP, NP hard . . . . .	13
2.20	Shannon . . . . .	13
2.21	Kolmogorov . . . . .	13
2.22	Solomonoff . . . . .	13

# 1 First Part of Semester

## 1.1 RAM Model

2 contexts:

1. Analysis: an idealised, simplified, abstract model to analyse complexity:
  - Hardware independant
  - Operations are equivalent:  $+, -, *, /, ==, <, >, \%$
  - Variables access, allocation or change
2. Turing complete language (with efficient access to data)

Question: Why do we need the RAM Model. Answer: Because otherwise Analysis too complex and no comparisons possible

## 1.2 Big O Notations

For the algorithmic analysis, we think about orders of magnitude with  $n$ , the problem size, is becoming very large.  $\mathcal{O}(n)$  means of ther order of magnitude of  $n$ , meaning  $10n, 100n, 0.1n$ . And so typical operations are:

- $\mathcal{O}(n) + \mathcal{O}(n) \approx \mathcal{O}(n)$
- $\mathcal{O}(n) + \mathcal{O}(n^2) \approx \mathcal{O}(n^2)$
- $\mathcal{O}(n) + \mathcal{O}(n \log n) \approx \mathcal{O}(n \log n)$

Question:  $\mathcal{O}(expn) + \mathcal{O}(n!)$  Answer:

## 1.3 Master Theorem and Derivation

Given recursion T using divide and conquer on a problem of size  $n$ , we write:

$$T(n) = aT(n/b) + \mathcal{O}(n^c),$$

$a$  is the number of branches in the recursion and  $b$  the reduction ( $a$  and  $b$  do not need to be 2 or the same!).  $\mathcal{O}(n^c)$  is the cost from work,  $aT(n/b)$  the cost for branching.

One can decompose the formula entirely, by  $\log_b n$  iterations, because  $\log_b n^1$  is the number of steps until the tree ends.

You can check:

$$T(n) = aT(n/b) + \mathcal{O}(n), \tag{1}$$

$$= a^2T(n/b^2) + a\mathcal{O}(n/b) + \mathcal{O}(n^c), \tag{2}$$

$$= a^3T(n/b^3) + a^2\mathcal{O}(n/b^2) + a\mathcal{O}(n^c/b) + \mathcal{O}(n^c), \tag{3}$$

$$= \dots \tag{4}$$

$$= a^{[\log_b n]}T(n/b^{[\log_b n]}) + \sum_{i=0}^{[\log_b n]-1} a^i \mathcal{O}(n^c/b^i). \tag{5}$$

---

<sup>1</sup>take nearest integer

To find a version close to the master theorem, one uses  $a^{\log_b n} = n^{\log_b a}$  and realise that  $T(n/b^{\lceil \log_b n \rceil}) = \mathcal{O}(1)$ , leading to:

$$T(n) \approx \mathcal{O}(n^{\log_b a}) + \sum_{i=0}^{\log_b n - 1} a^i \mathcal{O}(n^c / b^i).$$

For  $n \rightarrow \infty$ , we find 3 main cases:

- $\log_b a > c$ , branch costs dominate and  $\mathcal{O}(n^{\log_b a})$
- $\log_b a = c$ , most complicated combination:  $\mathcal{O}(n^{\log_b a} \log_b n)$
- $\log_b a < c$ , work costs dominate and simply:  $\mathcal{O}(n^c)$

Question: what is complexity of recursion:  $T(n) = 4T(n/2) + \mathcal{O}(n)$ , Answer:  $\mathcal{O}(n^2)$  Question: what is complexity of recursion:  $T(n) = 2T(n/2) + \mathcal{O}(n)$ , Answer:  $\mathcal{O}(n \log n)$

## 1.4 Master Theorem, Applications and Restriction

Restrictions: apply only on 1. recursions that 2. reduce a problem by a multiple...

- $T(n) = T(1.3n) + \dots$  no!
- $T(n) = T(n-1) + \dots$  no!

Question: what is complexity of "good" sorting algorithm?

Answer:  $T(n) = 2T(n/2) + \mathcal{O}(n)$ ,  $\mathcal{O}(n \log n)$ .

## 1.5 Multiplication Algorithms

Using divide and conquer

1. Standard multiplication in basis 10 (carry variable, divide and conquer and optimisation with Karatsuba)
2. Antic peasant multiplication (basis 2)
3. Matrix multiplication (Decompose into submatrices, optimisation with Strassen Method)
4. russian peasant Multiplication (not covered in course)

### 1.5.1 Standard multiplication in basis 10

Understand at least the code's ideas...

```
function multiplication(X, Y)
    # Multiply two non-negative integers represented as digit arrays (MSB first)
    # Returns the product as a digit array (MSB first)

    nX, nY = length(X), length(Y)
    if nX == 0 || nY == 0
        return Int[]
    end

    # Reverse to work LSB-first during computation
    x_rev = reverse(X)
    y_rev = reverse(Y)

    # Maximum possible length of product
    prod = zeros{Int, nX + nY}
```

```

# Convolution-style digit multiplication
for i in 1:nX
    for j in 1:nY
        prod[i + j - 1] += x_rev[i] * y_rev[j]
    end
end

# Normalize carries
carry = 0
for k in 1:length(prod)
    total = prod[k] + carry
    prod[k] = total % 10
    carry = total ÷ 10
end

# Remove leading zeros (from the back, since prod is LSB-first)
while length(prod) > 1 && prod[end] == 0
    pop!(prod)
end

# Return MSB-first
reverse(prod)
end

```

### Karatsuba:

Idea:  $a, b$  in  $\mathbf{Z}$ , can be decomposed as:

$a = a_0 \cdot 10^{n/2} + a_1$  and  $b = b_0 \cdot 10^{n/2} + b_1$ , and so we have few smaller multiplications:

$$a \cdot b = a_0 \cdot b_0 \cdot 10^n + (a_0 \cdot b_1 + a_1 \cdot b_0) \cdot 10^{n/2} + a_1 \cdot b_1,$$

and so we have  $T(n) \approx 4T(n/2)$ , that is by Master theorem  $\mathcal{O}(n^2)$ .

## 1.5.2 Matrix Multiplication

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B},$$

$$n \text{ operations } \forall i, j : c_{ij} = \sum_{k=0}^n a_{ik} \cdot b_{kj}.$$

Divide and conquer:

$$\begin{aligned}
 A &= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}, \\
 \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} &= \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.
 \end{aligned} \tag{4.4}$$

Decomposing with 8 sub-operations:  $T(n) = 8T(n/2) + \mathcal{O}(1)$ ,  
so  $\mathbf{T}(n) = \mathcal{O}(n^3)$  (master theorem  $c = 3 = \log_2(8)$ ).

Strassen optimisation  $\mathcal{O}(n^3)$

## 1.5.3 Paesant Multiplication

$a * b$  with  $a, b \in \mathbf{Z}$ ,

- $a == 0$ , then return 0,
- $a \% 2 == 0$ , return  $\lfloor a/2 \rfloor *_{paesant} b$ ,
- $a \% 2 \neq 0$ , return  $\lfloor a/2 \rfloor *_{paesant} b + b$ .

It is obviously a recursion.

## 1.6 Computations: n!, Fibonacci, ...

recursion easy to write but generate a lot of calls in the stack. Each time a function calls another one that calls another one, some space has to be allocated on a stack structure, in the memory. This behaviour is uncontrolled (how can the computer know the function size in certain cases and where to allocate best the function and data...) and can let to various problems, among other stack overflow.

Precision issue: For large numbers, the bits allocated to your value might not suffice, or the contributions of smaller numbers might be lost in approximations.

Alternative approaches are to compute using Dynamical computing or from ground up.

Question: How would you compute  $n!$  effectively, with large  $n$ ?

Answer: With a for loop... and starting from low to high. Using also a long list to be gradually populated as number grows (+divide and conquer...)

Question: Efficient computations of Fibonacci Answer: Either from bottom up with space complexity of  $\mathcal{O}(1)$  (two pointers) and time complexity of  $\mathcal{O}(n)$  or With a Matrix as in exo 3, 1: as  $\mathcal{O}(\log_2 n)$

## 1.7 Knapsack 0–1

Definition: bag of size  $W$ , with a number of objects  $i$  of sizes  $w_i$  smaller than  $W$  and some price  $V_i$ . Optimise the price within the bag.

Not greedy because as more objets are placed in the bag (filling from bottom up), the optimal strategy can totally change.

Question: How to solve Knapsack 0–1:

Answer: Bottom up tabulation (like in course), set  $ks[i][w] = 0$  maximum value achievable using the first  $i$  items with capacity  $w$ , set up to 0.

Iteration:

$$ks[i][w] = \begin{cases} ks[i-1][w], & \text{if } w_i > w, \\ \max \left( ks[i-1][w], ks[i-1][w - w_i] + v_i \right), & \text{if } w_i \leq w. \end{cases}$$

## 1.8 Merge Sort

Divide and conquer on array for sorting.

Question: time complexity?

Answer:  $T(n) = 2T(n/2) + O(n)$

By master theorem:  $\mathcal{O}(n \cdot \log n)$  Question: write pseudocode for merge sort

Answer:

```
1: function MERGESORT( $A, p, r$ )
2:   if  $p < r$  then
3:      $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$ 
4:     MERGESORT( $A, p, q$ )
5:     MERGESORT( $A, q + 1, r$ )
6:     MERGE( $A, p, q, r$ )
7:   end if
8: end function
```

## 1.9 Quick Sort

Use a pivot to sort around.

Question: write a pseudocode:

```
1: function QUICKSORT( $A, p, r$ )
2:   if  $p < r$  then
3:      $q \leftarrow \text{PARTITION}(A, p, r)$ 
4:     QUICKSORT( $A, p, q - 1$ )
5:     QUICKSORT( $A, q + 1, r$ )
6:   end if
7: end function
```

Question: Time worse and best time complexity and explain

Answer: worse  $O(n^2)$ , best  $O(n \cdot \log n)$ . Problem is that the divide and conquer method can be very unbalanced. So the recursion is going to oscillate between:

$T(n) \sim 2T(n/2) + O(n)$  and  $T(n) \sim 2T(n-1) + O(n)$  (master thm not applicable!) and so resp.  $O(n \log n)$  and  $O(n^2)$ .

Technique to improve unbalance problem is for instance averaged pivots.

## 1.10 Heap Sort

Array  $\longleftrightarrow$  Complete Binary Tree. Sorts in-place — no extra memory needed!

Pointers, root: 0:

- Parent( $i$ )  $\rightarrow \lfloor \frac{i}{2} \rfloor$
- Left( $i$ )  $\rightarrow 2i$
- Right( $i$ )  $\rightarrow 2i + 1$

Question: What are steps to sort arrays?

Answer: alternate heapify and swap

Question: what is max heap property?

Answer: for every node in a binary tree, the value of the node must be greater than or equal to the values of its children

Question: time complexity and space complexity and how you compute them?

Answer: space complexity is simply  $O(n)$  because everything done in place

time complexity:  $O(n \cdot \log n)$ , because  $\log_2 n$  operation at most in heapify, performed at most  $n$  times.

## 1.11 Dynamical Computing

Question: When Dynamical Computing applies

Answer: When Problem has Optimal substructures and overlapping subproblems

Question: What are optimal substructures

Answer: A problem has Optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems

Question: Overlapping Subproblem?

Answer: Overlapping problem when solution executing multiple time same operation. Question:

Typical approaches with dynamical programming:

Answer: Memoisation and tabulation, bottom up

Question: Examples of Dynamical Computing  
Fibonacci numbers, shortest paths, knapsack problem, edit distance.

## 1.12 Greedy Computing

Requirements:

1. **greedy-choice property:**  
globally optimal solution  $\Leftrightarrow$  local optimal (greedy) choices
2. **optimal substructure**

Example where greedy algorithm suboptimal:

- life!?
- road...
- 0-1 knapsack problem

Example where used:

- Huffman encoding
- distances on graph (dijkstra)
- Minimum spanning tree with Kruskal
- graph colouring

## 1.13 Edit (Levenshtein) Distance

It is a dynamical computing approach to define a distance between strings. It can be easier understood with the course, part IV or 4.

Question: what are the operations allowed

Answer: edit, remove and add letter.

Question: what is the edit distance between cat and cut, Saturday and Sunday. Answer: 1, 3

## 1.14 Explain Divide and Conquer

Main ideas: divide, and then do something, then merge

# 2 Second Part of Semester

## 2.1 Course Allocation Problem

Method is simple: first sort courses by ending time and fill the courses by choosing the first or next first course compatible (with starting time  $\leq$  previous end time).

Once this sorting occurred, just run greedy search on these:



1. choose first course of above sorted list
2. assume you go to course until it finishes
3. take next course starting after last course ended

One can show that this simple strategy is optimal by contradiction, comparing greedy search with optimal and showing greedy search has to be optimal.

## 2.2 Huffman Encoding

Greedy algorithm for text data compression.

Question: How is encoding performed  
 counters are created for all characters (if encoding at character level).  
 Then a greedy bottom up strategy is used to generate a tree representing the data with bits.

Question: How to compute loss for Huffman encoding  
 One can use Shannon entropy and compare it with the average encoding length:

$$L = \sum_c p_c l_c,$$

with a sum over all characters  $c$  and  $l_c$  the length of the encoding and  $p_c$  its probability of the fraction of the data that is  $c$ .

One can find that both averaged length  $L$  and the Shannon Entropy are close, showing that two views of complexity (entropy versus minimal length) are close and that Huffman encoding, despite its relative simplicity, is very efficient.

## 2.3 Graph

$$\mathcal{G}(V, E).$$

- All main definitions in the course...
- vertex, edge, directed, undirected graphs.
- forest, tree
- Walk, Path, reachable, connected
- cycle, acyclic graph
- strongly connected
- directed Acyclic graph (DAG!)
- weighted, unweighted graphs

## 2.4 Tree

- Tree, spanning Tree, forest

## 2.5 Tree Representation

Adjacency Matrices, Sparse Adjacency Matrices  $\mathcal{O}(V)$   
Review CSR, CSC, COO.

## 2.6 Graph Operations

Union, Intersection, Join

## 2.7 Operations on Graphs

Depth First Search, Breadth first search using resp. stack and queue (why?). Resp. First In First Out (FIFO) or Last In First Out (LIFO). Complexity is  $\mathcal{O}(V + E)$

Question: Why is time complexity  $\mathcal{O}(V + E)$ ? Answer: visiting  $V$  vertices/nodes and  $E$  edges...

## 2.8 Advanced Concepts

- Clique
- Minimum Spanning Tree
- Shortest Path
- Graph Colouring
- Planar Graphs
- Strongly Connected Component

## 2.9 Transformations of Graphs

inverse, transpose and dual

## 2.10 Algorithm: Topological Sort

Using BFS... Simple algo. Know how to do it.

Question: Given this graph image, can you sort it topologically? Answer: see internet or examples... notice that we have looked at sorting with DFS and not BFS, that leads to different, correct results...

Question: has Topological sorting unique solution

Answer: Not at all, depends on starting point

Question: Application of topological sort?

Answer: dependency trees (modules), Equivalence circuits linear code...

## 2.11 Algorithm: Cycle Detection

Know basic simpler idea, DFS until coming back. Understand why it is not a serious solution in general.

Review basic setup of Bron Kerbosch. It is enough to have run the algorithm on examples in case you are asked to run it on an image. No need to know algorithm.

## 2.12 Find Strong Components

Kosaraju: consider a directed graph  $G$ .

1. empty stack  $S$ .
2. start from a random node and follow the vertices with DFS, pushing the vertices into the stack.
3. transpose  $G$  (invert all edges)
4. run DFS from the top of the stack (last visited)
5. When/if hitting an element lower in the stack, that is a SCC
6. remove all elements from SCC from the stack
7. continue

Question: What is Kosaraju time complexity  $O(E + V)$ ? Answer: Because at most running 2 DFS, of complexity  $O(E + V)$ .

## 2.13 Cut Theorem

## 2.14 Find Minimum Spanning Tree

### 2.14.1 Cut Property / Theorem

For any cut  $(S, V - S)$ , if edge  $e$ 's weight is the minimum among all edges crossing the cut, then  $e$  belongs to some MST of  $G$ .

Concrete applications are:

### 2.14.2 Kruskal

Greedy approach: "Always pick the smallest available edge that doesn't create a cycle"

1. sort all edges by weight
2. connect nodes from the smallest edge and build MST
3. if edge not creating cycle, add to MST otherwise go to next larger edge

### 2.14.3 Jarnik Algorithm

Build MST from ground up

1. start at random vertex/node (initial MST)
2. find vertices around initial MST
3. add minimum vertex to MST and connect a new point
4. list new vertices around MST
5. select the minimum to add it to MST and connect new point

6. ...

7. remove edges not belonging to MST

Question: Efficient implementations of Jarnik Algo.?

Answer: using minheap to track the MST nodes and their edges

Question: What is the time complexity and why?

Answer:  $O(E \log V)$ ,  $\log V$  for the maximum tree length for  $V$  nodes and there are  $E$  edges to visit.,

## 2.15 Graph Colouring

Question: What is the Chromatic number?

Answer: the number of colour to have no adjacent node of similar colour.

Question: How hard is it to find the chromatic number?

Answer: NP hard.

Question: Examples of Graph Colouring problems?

Answer: Among other Sudoku..., register allocation in compilers, scheduling, ...

### 2.15.1 Greedy Algorithm

Simplest possible algorithm:

1. start on some node
2. colour it
3. go to next neighbour with BFS and colour it with other colour
4. try to keep colour minimal if you can

## 2.16 Dijkstra Algorithm

Find minimum distance...

1. Set distance to source = 0. Set all other distances to  $\infty$ . Mark all nodes unvisited.
2. While there are unvisited nodes
3. Choose the unvisited node with the smallest known distance
4. For each neighbor of that node:
5. (a) Add the edge weight to the current node's distance.  
(b) If this gives a shorter path to the neighbor, update its distance.  
(c) Mark the current node as visited.

Question: what is a well known limitation of the Dijkstra algorithm?

Answer: Dijkstra algorithm can not handle well negative weight and risks to enter infinite, negative loops when facing them.

## **2.17 Finite Functions**

Nand equivalence with AND, OR, NOT.  
Equivalence Circuits, straight line programs  
Finite Automata

## **2.18 Turing**

## **2.19 P, NP, NP hard**

## **2.20 Shannon**

## **2.21 Kolmogorov**

## **2.22 Solomonoff**