Содержание

Algorithms Course Proposed Schedule (Academic Year 2025–2026)

[Jair Wuilloud]

September 2025

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:

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

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:

$$\begin{aligned} T(n) &= aT(n/b) + \mathcal{O}(n), && (1)\\ &= a^2T(n/b^2) + a\mathcal{O}(n/b) + \mathcal{O}(n^c), && (2)\\ &= a^3T(n/b^3) + a^2\mathcal{O}(n/b^2) + a\mathcal{O}(n^c/b) + \mathcal{O}(n^c), && (3)\\ &= ... && (4)\\ &= a^{[log_b n]}T(n/b^{[log_b n]}) + \textstyle\sum_{i=0}^{[log_b n]-1} a^i\mathcal{O}(n^c/b^i). && (5) \end{aligned}$$

To find a version close to the master theorem, one uses $a^{log_b n} = n^{log_b n}$ and realise that $T(n/b^{[log_b n]}) = \mathcal{O}(1)$, 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 $n \to \infty$, 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: $\mathcal{O}(n^{log_b a} log_b n s)$
- $log_b a < c$, work costs dominate and simply: $\mathcal{O}(n^c)$

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

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

[1] take nearest integer

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 paesant multiplication (basis 2)
3. Matrix multiplication (Decompose into submatrices, optimisation with Strassen Method)
4. russian paesant 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 $a = b_0 \cdot 10^{n/2} + b_1$, 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 operations $\forall i,\ j:$ $\quad c_{ij} = \sum_{k=0}^{n} a_{ik} \cdot b_{kj}.$

Divide and conquer:

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

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}. \tag{4.4}$$

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:

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

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: $\mathcal{O}(n \cdot logn)$ Question: write pseudocode for merge sort
Answer:

```
1: function MergeSort(A, p, r)
2:     if p < r then
3:         q ← ⌊(p+r)/2⌋
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 ← 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 $\mathcal{O}(n \cdot logn)$. 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. $O(nlogn)$ 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 $\mathcal{O}(n)$ because everything done in place
time complexity: $\mathcal{O}(n \cdot logn)$, 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 (dijskra)
- 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 understund 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 Cours 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 <= previous end time).

2.2 Huffman Encoding

Greedy algorithm for text data compression.
Question: How is encoding performed
Question: How to compute loss for Huffman encoding

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
- weighted, unweighted graphs

2.4 Tree

- Tree, spanning Tree, forest

2.5 Tree Representation

Adjacency Matrices, Sparse Adjacency Matrices $\mathcal{O}(V)$

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 $\mathcal{O}(V + E)$

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.

2.11 Algorithm: Cycle Detection

Know basic idea or basic setup of Bron Kerbosch

2.12 Find Strong Components

Kosaraju uses a stack and transpose. Maybe you remember this one.

What is search and why like DFS? Because at most a multiple of DFS.

2.13 Cut Theorem

2.14 Find Minimum Spanning Tree

Kruskal and Jarniks

2.15 Graph Colouring

Greedy algo

2.16 Dijskra Algorithm

2.17 Finite Functions

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

2.18 Turing

2.19 P,NP, NP hard

2.20 Shannon

2.21 Kolmogorov

2.22 Solomonoff