Algorithms Fundamentals

The Halting Problem and Computational Models

Let *h* be the function that establishes whether any program p executed with inputs i ends or run forever.

For any *f(a,b)*, we define a *gf(i)* such that it returns 0 if *f(i,i) = 0* and infinity otherwise.

The Halting problem is a computer science counterpart of Godel's Theorem.

The **Church-Turing Thesis** states that every effectively calculable function is computable, i.e. there exists an implementation for any given algorithm. For this reason, we can infer that all reasonable computational models (which allow some standard minor operations, such as assignment, looping, etc.) are equivalent and thus we can use the easiest ones for our purposes.

Our computational model (The **Random Access Machine, RAM**) includes variables to store data, arrays, int and float constants, algebraic functions, assignment, pointers (without pointer arithmetic), conditional and loop statements, procedures and functions. However, the RAM model doesn't have the hardware limitations, memory hierarchy and execution time that we find in real hardware, since it represents a generalization of the latter.

Time Complexity

To measure the efficiency of a program, we may use **execution time**. For algorithms this is not a realistic estimation since it relies heavily on hardware and context (language, compiles, memory handling, CPU, size of inputs, etc.).

For algorithms, we may use **scalability** instead, since it measures the capacity of a system of handling input growth for a range of inputs, estimating the relation between input size and execution time.

In order to estimate scalability, we are simply looking at **asymptotic behaviour**, excluding constants and using the big O notation.

If we have a constant and an input value for a function *g(n)* such that g is <= than a *f(n)* for all values of n after the input value, then we say that *g(n)* belongs to O(f(n)). O notation sets an upper bound for the growth of complexity in a program.

Big omega notation is the same concept, but in the best case scenario when f shrinks instead of when g grows, setting a lower bound. Finally, big Theta notation is the intersection between the two previous concepts, defining the exact behaviour of a function.

Other Notions

Abstract Data Types

Arrays, Single Linked Lists, Double Linked Lists, Queues (FIFO), Stacks (LIFO).

Graphs

Nodes, edges, directed and undirected, path as sequence of edges, connected and acyclic graphs. A connected, acyclic graph is called tree. If a node is selected to be the root, the tree becomes a data structure.

The depth of a node is its distance from the root in term of edges. All the nodes having the same depth are said to be on the same level. Parent, children and siblings nodes. Leaves and internal nodes. Height as max depth among leaves. N-ary trees with n children, complete if all the internal nodes have n children.

Matrix Multiplication

The naïve implementation of the matrix multiplication algorithm has complexity $O(n^3)$.

To make it more efficient, we may apply a **Divide and Conquer** strategy by splitting the two squared matrices in k blocks of size s. The value of the block C_{ij} will be given by:

$$C_{ij} = \Sigma_{l=0}^{k} A_{il} \times B_{lj}$$

The products can be computed recursively, by splitting subsequent blocks in more blocks using the basic recursive algorithm that splits a matrix in four blocks. At each level of recursive splitting, we are dividing by two the size of the matrix on which we are operating.

Since each block C_{ij} for a block requires eight multiplications, the complexity at the i-th level of recursion is

$$T_M(n) = c(\frac{n}{2^i})^2 \times 8^i = c(n^2) \times 2^i$$

Since we are performing binary splitting, the height of the recursion tree is $log_2(n)$

$$T_M(n) = \Sigma_{i=0}^{log_2(n)} cn^2 2^i$$

$$= cn^2 \Sigma_{i=0}^{log_2(n)} 2^i$$

$$= cn^2 \frac{2^{1+log_2(n)} - 1}{2 - 1}$$

$$= cn^2 (2^{(1+log_2(n))} - 1)$$

$$= cn^2 \times 2 \times 2^{log_2(n)} - cn^2$$

$$= 2cn^3 - cn^2 \in \Theta(n^3)$$

This doesn't held better complexity than the previous algorithm, but can be reformulated using **Strassen's Algorithm** in order to reduce the number of multiplications (recursive calls) and increasing the number of sums, which are anyways of fixed complexity $\Theta(n^2)$.

The new complexity equation is:

$$T_M(n) = 7 \times T_M(\frac{n}{2}) + \Theta(n^2)$$

since the products have been reduced to 7. The final asymptotic complexity of Strassen algorithm is $\in \Theta(n^{log_2 7}) \approx \Theta(n^{2.81})$.

The algorithm is not in-place, a.k.a. it requires a non-constant amount of additional memory. This shows how a careful handling of the auxiliary memory may make the difference in implementation, simply by reducing the number of recursive calls.

Chain Matrix Multiplication

We want to reduce the total number of scalar product performed for the multiplication of a chain of given matrices $A_1, A_2, \ldots, A_n$ through applying optimal parenthesizations.

The ideal way to perform this is to recursively compute optimal parenthesizations and use dynamic programming to avoid the number of possible parenthesization to explode.

We recursively compute the product of all matrices between A_i and A_j by assuming that the optimal split is $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$ with k ranging from i to $j-1$ and summing the recursively computed product between A_i and A_k, between A_{k+1} and A_j and the total number of scalar products needed to compute the product between the two recursive subproblems (represented by the dimensions of the resulting matrices p_{i-1}, p_k and p_j). The product of those matrices is then minimized w.r.t. k. The stopping condition is that the product is 0 when $i = j$, which represent the case in which we are simply trying to compute the product between two matrices, in which the number of scalar product is given simply by the scalar products between the two matrices (no recursive subproblems).

We use a matrix m to store the number of multiplication computed that way and a matrix s to store the k that minimizes the number of multiplication for the current subproblem (e.g. k = 3 means that we obtain the best parenthesization by placing a parenthesis after the third element of the chain).

Inside m, in position (i, j) i will find the minimal number of scalar multiplications needed to solve the chain matrix multiplication between matrices A_i and A_j of the original chain $A_1 \ldots A_i \ldots A_j \ldots A_n$.

By performing the same task iteratively we can avoid recursion and thus avoid the memory required by each step.

The complexity of the algorithm is $\Theta(n^3)$, where n is the size of the chain. This result is much better than the original complexity to perform matrix multiplication, which is $\Theta(2^n)$.

Searching and Sorting

Complexity in unsorted array: at least $O(n)$ (scan all elements).

Complexity in sorted array: $O(logn)$ with dichotomic search.

Insertion sort

Switch places when right elements is smaller than left element, iterating through the whole array. Complexity: $O(n^2)$ (n to iterate through the array, n to switch elements in the worst case).

Quick sort

Select pivot, split array in two subarrays (with all elements respectively smaller and greater than pivot) and recursively apply the same procedure on these subarrays.

Complexity to perform the splitting in subarrays: $O(n)$. Can be done in place (see slides).

Worst case total complexity: $O(n^2)$ if one of the subarrays is always empty since it becomes $T_Q(n) = T_q(n-1) + \Theta(n)$.

Best case: When the splitting procedure produces two arrays which have constant ratio, for all subtree we have to perform n operations for the splitting and the number of subtrees approaches $logn$, so the complexity is $\Theta(n\ log\ n)$ ($O(n\ log\ n) = \Omega(n\ log\ n)$).

Heapsort

Any sorting algorithm using comparisons can be modeled as a decision-tree model, with a number of leaves (that represent all possible permutations of elements in the array to be sorted) equal to $n!$ and thus the minimum complexity for the worst case scenario in any case is $O(n \log n)$.

By introducing ad-hoc constraints such as a bounded domain and a uniform distribution of the array values, we can achieve sorting in linear time.

Select

We can find the ith element in an unsorted array by applying the partition algorithm we used in quicksort to partition. In the worst case (the splitting between values smaller and greater than the pivot is unbalanced) the complexity is $O(n^2)$. To avoid this, we should choose optimally the pivot by splitting the array in chunks and use as pivot the median of medians of all chunks. In this case, the complexity becomes, for example:

$$T_S(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T_S(\lceil n/5 \rceil) + T_S(7n/10 + 6) + \Theta(n) & \text{otherwise} \end{cases}$$

for a vector splitted in 5 chunks, where $T_S(n/5)$ is the complexity of sorting each chunk and $T_S(7n/10 + 6) + \Theta(n)$ is an upper bound for the number of elements smaller or equall to the selected pivot m.

The algorithm is optimal for chunk size of 5, and is useful to upper bound the number of operations.

```
DEF SELECT_PIVOT(A, l = 1, c=|A|)
    FOR i <- 0 UP TO (r-l)/5
        INSERTION_SORT(A, 5*i+l, MIN(5*i+l+4, r))
    ENDFOR

    FOR i <- 0 UPTO (r-l)/5
        SWAP(A, l+i, MAX(5*i+l+2), r))
    ENDFOR
    RETURN SELECT(A, ((r-l)/5 + 1)/2, l , l+ (r-l)/5)
ENDDEF
```

Heaps

A heap is a data structure totally ordered w.r.t a relation $\preceq$ (a generalization of comparison, can be $\leq$ in min-heaps and $\geq$ in max-heaps). They are used to implement priority queues.

Binary heaps are usually represented as trees, which can be converted from arrays (see slides). The characterizing relation, called **heap property**, is valid only hierarchically and not between siblings nodes (as opposed to binary trees, where we have that the rightmost node is > than the leftmost on the same level).

In the Heapify method presented in slides, $\preceq$ represent the relation of the heap and there is an error in the comparison: it is `H[j] ⪯ H[m]` instead of `H[j] ⪯ m`.

- Binary heap has height $log_2\ n$
- Level l contains 2^l nodes at most.
- Each call at heapify cost at most $log_2\ n$.
- Thus, the total cost of building a heap has a cost of **at most** $\Sigma_{l=0}^{log_2 n} 2^l \times c \times \log_2 n = n\ log_2\ n \in O(n\ log\ n)$
- The number of nodes having height h is $\left[\frac{n}{2^{h+1}}\right]$ at most.
- Heapify perform at most $c * h$ operations:

$$\Sigma_{h=0}^{log_2 n} \frac{n}{2^{h+1}} * c * h \leq cn\Sigma_{h=0}^{\infty} \frac{h}{2^h} = cn\frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2 * c * n \in O(n)$$

Since $\Sigma_{h=0}^{\infty} \frac{h}{2^h} = \frac{a}{(1-a)^2}$ when $a < 1$ and we used $a = \frac{1}{2}$ since it is a binary heap.

Decreasing a key w.r.t. $\preceq$: Complexity

While heapify proceeds from the root to the leaves of the heap,
the procedure to preserve the heap property after decreasing a
key value w.r.t. $\preceq$ (increase for $\geq$, decrease for $\leq$) moves the problem up to the root by performing switches.

Since the complexity of a switch is constant and the height of the tree is $\log n$, the total complexity is $O(\log n)$.

```
DEF DecreaseKey(H, i, value):
    IF H[i] <= value:
        ERROR("I can only decrease key values")
    ENDIF
    H[i] <- value
    WHILE NOT (IsRoot(i) OR H[Parent(i)] <= H[i])
        Swap(H, i, Parent(i))
        i <- Parent(i)
    ENDWHILE
ENDDEF
```

Heap Sort

Heap sort has complexity $O(n \log n)$, but the creation of the heap with complexity $\Theta(n)$ should be also taken into account.

Dynamic indexes

In a setting where the data present in a specific context is going to change often, we need a new data structure to perform the addition, the removal and the querying in the most efficient way (e.g. without having to pass through the whole structure to reorganize data after addition/deletion).

Binary Search Trees (BST)

The tree is made by nodes having a **key** and a **value**, a **left and right child** (which will be set to NIL if they are not present), and possibly a **parent node**. The tree itself should have a **root** node, which is connected to all others hierarchically.

Removing a node from BST:

```
DEF Transplant(T,x,y)
    IF x.Parent = NIL
        T.Root <- y
    ENDIF
    y.Parent <- x.Parent
    IF IsRightChild(x)
        y.Parent.Right <- y
    ELSE
        y.Parent.Left <- y
    ENDIF
ENDDEF
```

The strategy to delete a node with two children is to proceed in its right subtree to find the leftmost child (at most a child on the right), moving this node in the place of the one we want to delete and moving up the whole subtree from where the node has been moved.

BSTs without balancing may perform at worst as linked lists. However, balancing is an expensive operation if it's done globally. Thus, we may want to try and balance only the unbalanced part of the tree.

Red-Black Trees

All nodes are either red or black. Root is black, all leaves (NIL) are black and all red nodes must have black children. Branches starting from any node must have the same number of black children.

Any RBT with n internal nodes has height of at most $2\log_2(n+1)$ nodes. The ratio between the height of two branches should be at most 2.

Graphs

In the depth-first search, numbers on the nodes represent the finding time and finishing time of the node. When a node isn't connected to any white node, the backtracking moves back in the graph of a node and colors the node black.

A **Strongly Connected Component (SCC)** is a sub-graph s.t. for every pair of nodes in it, there is a path from one to the other and vice versa (no path = zero-length path = each node is a basic SCC by itself). To identify them, we may use Depth-First Search to spot loops. The **Minimum discovery time (lowlink)** of a subtree is the minimum among the discovery times in the subtree. If the discovery time of a child node in a subtree is greater than the one of its parent, we have that there must be either a back edge connecting them, or a forward edge connecting to a different SCC.

The discovery time and the lowlink of the first node to be visited by DFS in a SCC are intuitively the same. All the nodes having discovery time higher than their own lowlink belong to the same SCC as the node having that lowlink as discovery time.

If we perform the DFS traversal and list all nodes in order of finishing time (from lower to higher), we can use a stack to iteratively spot SCCs, removing them from the stack.

Shortest Paths

We used BFS to build the tree of shortest paths from a node to all other nodes on unweighted (plain) graphs. The time required to compute the shortest path in that setting was $O(|V| + |E|)$, and the processing of grey nodes was handled by a FIFO queue.

In the case where a node is close to source but with a high weight label on its edge, while another longer path that connects it to source has total label weight which is smaller, we see that raw BFS is not apt for shortest pathfinding in weighted graphs.

Instead, we want to pre-label our nodes with a candidate distance while they are grey, and each time another smaller candidate distance is computed for that node (coming from a new path), it replaces the old one until we get the final distance. At each step we extract from the queue the node with smallest distance from the source and finalize it. This is known as **Dijkstra's Algorithm**.

The Dijkstra's Algorithm has time complexity $O(|V| + |E| + \log|V|)$ when a Fibonacci heap is used to store the nodes and works **only for edges having positive labels**.

How to compute all shortest paths between all nodes in a graph? A possible approach would be to run Dijkstra for all nodes, but there are more efficient ways to do so.

In the Floyd-Warshall's algorithm, we have three matrices:

- The **adjacency matrix**, which represent the weights of each connection between nodes (row represent the from-node index, column the to-node index)
- The $\Pi^{(n)}$ **matrix** is the matrix containing all the predecessors for each shortest path at step n. At the first step, it contains all the predecessors for all the edges in graph (NULL when there is no edge, row_id when there is a connection).
- The $D^{(n)}$ **matrix** that contains the minimum lenghts all nodes in the graph (row represent the from-node index, column the to-node index)

The complexity of Floyd-Warshall is $\Theta(|V|^3)$ while the Dijkstra-based approach implemented with a Fibonacci heap has complexity $O(|V|(|V|\log(|V|+|E|)$, which is cubic w.r.t. edges only in the worst case scenario. The reason to use Floyd-Warshall then is to deal with negative-weighted paths, which Dijkstra's algorithm cannot manage.

Routing

The routing problem is similar to the **Single Source Shortest Path (SSSP)** problem: instead of using a single source to find shortest path to all destinations, we use all possible paths to find the shortest way from a source to a single destination.

We see that plain Dijkstra is not optimal since it looks in all directions We can use a light version of the Dijkstra's algorithm (called *A algorithm**to perform routing: the best complexity will be the same as for SSSP.

Since to compute shortest paths in real life we also have to take in account the topology of the territory connecting the two locations, the A* algorithm must take in account a heuristic distance which can be whichever fitting distance (Euclidean, Manhattan, Chebyshev).

This approach allows to reduce the number of queue extractions performed by the Dijkstra's algorithm dramatically.

Routing techniques can be applied to many fields:

- Routing of internet packets between servers
- Parcel delivers
- Travelers commuting

String-matching

An **alphabet** Σ is a set of symbols e.g. $\{0, 1, a, \ldots, z\}$

A **string** is a sequence of symbols contained in an alphabet.

Σ^* is the set of all strings built on alphabet Σ, including the empty string ϵ.

Two strings $x, y \in \Sigma^*$ can be concatenated as xy.

If two strings are both prefixes or suffixes of the same string and one is smaller than the other, then one is necessarily the prefix/suffix of the other.

We can say that a string P (pattern) occurs inside a longer string T (text) if we can find a slice of T such that:

$$T[s+1 \ldots s+m] = P$$

Where s is the **shift** (positive integer) for P in T and m is the length of P. A shift is called **valid** if the aforementioned equation is verified.

The string-matching problems requires to find all the valid shifts for a pattern P inside a text T.

The easiest way to solve the problem is to proceed iteratively by selecting a shift, trying to match the characters when possible.

```
DEF Naive_PM(T, P)
    Valid <- []
    FOR S <- 0 UP TO |T| - |P|:
        q <- 1
        WHILE q < |P| AND T[s+q] = P[q]:
            q <- q+1
        END WHILE
        IF q > |P| THEN # We matched all characters
            Valid.APPEND(S)
        ENDIF
    ENDFOR
ENDDEF
```

The complexity is $O(|P| * |T|)$, which may be fine for small patterns but for large ones may be inefficient.

The Knuth-Morris-Pratt Algorithm

A better idea is to look for repetition inside prefixes and suffixes of the pattern. We can exploit repetition inside the pattern in order to reduce the number of matchings.

E.g. Let's assume to have matched $P_5 = [a, b, a, b, a]$, where $P_3 = [a, b, a]$ is the largest pattern to be both prefix and suffix of the pattern. For P_3 the largest pattern to satisfy the same condition is $P_1 = [a]$, while for P_1 it is ϵ.

In this process, we are finding $\pi^*[q] = \{\pi[q], \pi^2[q], \dots, \pi^t[q]\}$ (the set of all prefixes of P_q that are also suffixes for it.)

Knowing that if a $\pi[q] > 0$ then $\pi[q] - 1 \in \pi^*[q-1]$, where $\pi^*[q]$ is the result of the **Prefix-function iteration lemma**, the set of all the prefixes of P_q.

We then have that:

$$\begin{array}{ll} \pi[q] = 0 & \text{if} \quad E_{q-1} = 0 \\ \pi[q] = 1 + \max\{k \in E_{q-1}\} & \text{otherwise} \end{array}$$

A pseudocode version of the algorithm:

```
DEF Compute_Prefix_Function(P)
    Pi <- INIT_ARRAY(|P|)
    Pi[1] <- 0
    k <- 0
    FOR q <- 2 UP TO |P|
        WHILE k > 0 AND P[k + 1] != P[q]
            k <- Pi[k]
        ENDWHILE
        IF P[k + 1] = P[q]
            k += 1
        ENDIF
        Pi[q] <- k
    ENDFOR
    RETURN Pi
ENDDEF
```

The prefix function is used to skip directly to the shift that can match at least the length of the smallest prefix-suffix of the pattern matched

Since the number of executions of the while loop is bounded by the number of times k gets increased, we have that the complexity is equal to $\Theta(|P| + |T|)$. The complexity of the prefix function is $\Theta(|P|)$

The Boyer-Moore-Galil Algorithm

The algorithm is different from the others because it performs the matching backwards. The algorithm was developed originally by Boyer and Moore having a complexity that is worse than KMP in worst case, until Galil added his rule to improve the worst case complexity. It exploits three main ingredients:

- **Good-suffix rule**: When we try to match our pattern (backwards) and there is a mismatch, we move to the left (the rightmost after the current one) inside the pattern to find another occurrence of the part that was already matched that is **preceded by a different character** or at the beginning of the pattern and try to match the newly chosen pattern backwards. Since it is basically the inverse of the prefix function, its complexity is still $\Theta(|P|)$.
- **Bad-character rule**: In addition to the good-suffix rule, when we are trying to perform the matching and there is a mismatch, we move to the left (the rightmost after the current one) inside the pattern to find the next occurrence of the character that was mismatched. If no occurrence is found, we can shift the pattern matching procedure to the character directly after the not-found character, since we know for sure that no pattern will match it. The complexity of this procedure is $\Theta(|P| + |\Sigma|)$, where Σ is the alphabet used to build T.
- **Galil's rule**: If a match has been discovered and P is k-periodic (not necessarily complete, e.g. ABABA is a valid 2-periodic incomplete patter), we can shift our pattern forward by k and avoid to match all characters that were already matched in the previous step (we avoid a total of $|P| - k$ comparisons). This can be done in linear time w.r.t. the pattern length, aka $\Theta(|P|)$

We proceed as follows:

- Try to match P on T backwards with initial shift.
- If mismatch, select the largest shift among those suggested by the good-suffix and the bad-character rules.
- If a valid shift is found, apply Galil's rules.

This gives us a worst case of $\Theta(|P| + |T|)$, but most of the time in sublinear time.

Multiple Patterns String Matching

We can use Boyer-Moore-Galil to try a multiple pattern matching (all subpatterns of a pattern), but the complexity of the approach is $O(|T| * \sum_i |P_i|)$, which isn't really optimal.

A more interesting approach is given by a tree-based solution, in which the tree contains all the paths for all possible combinations of subpatterns, having nodes with values equal to the shift needed to match the subpattern inside the pattern.

While this approach is linear w.r.t. patterns, the hard part is building a tree that conforms to the previously specified requirements.

The **active point** is the first node in the boundary path (following up the prefix functions from the longest path) that is NOT a leaf in the tree (in the example, the first left C is the active point since remounting through the prefix functions in red from the last leftmost node, it is the first one that is not a leaf). The **endpoint** is the first node coming before the active point having the added character as direct descendant.

The complexity of building a Suffix Trie is $\Theta(\Sigma(T))$, where Σ is the set of all substrings of T. We have that $|\Sigma(T)| \in O(|T|^2) \rightarrow O(|T|^2)$. This cost can be reduced by reducing redundancy.

Canonize steps take $O(|T|)$, while the steps to build the tree after canonizing take $\Theta(|T|)$, thus in total building the tree takes a complexity of $\Theta(n)$. This brings the total complexity for multiple pattern matching to $\Theta(|T|) + \Theta(\sum_i |P_i|)$, and the tree takes space $\Theta(|T|)$. However, this approach handles only one of the pattern valid shifts and may require up to $(2+1+|\Sigma|) * |T|$ words of RAM, which can be way too much for tasks such as genome sequencing.

An approach used to fix this is **Suffix Array**, where arrays are used to store suffixes in lexicographical order and can be searched with dichotomic search.