AC Notes

Daniel Nykjær, Ivik Hostrup, Patrick Østergaard

SW19 AC Exam Notes 2022

Contents

1 Greedy algorithms

1.1 Greedy properties

A problem can be optimally solved using a greedy approach if one can prove that the problem has two properties.

1.1.1 Greedy choice property

This property ensure that we can build a globally optimal solution by making locally optimal (greedy) choices. In other words, at each subproblem where we need to make a choice, we need only consider what looks best in that specific subproblem. We do not consider any other subproblems.

1.1.2 Optimal substructure

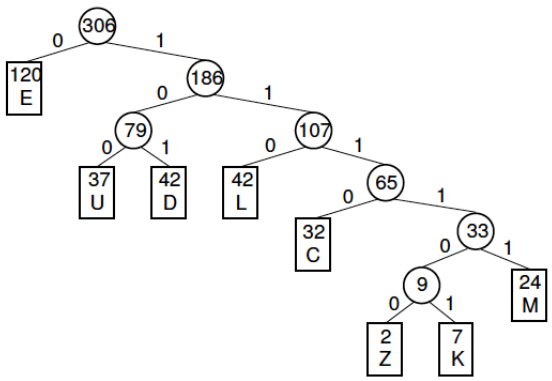
A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the sub-problems. In other words, a problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblem.

1.2 Huffman coding

Huffman codes compresses data efficiently. For each character in a file, we can either use a fixed-length code to identify each character or a variable-length code.

	a	b	c	d	e	f
Frequency	45k	13k	12k	16k	9k	5k
Fixed-Length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1101

The fixed-length code would result in all characters using the same amount of space. Therefore it would be best to use a variable-length code for each character where the most frequent letter has the shortest code. This saves a lot more space than using fixed-length.
In a text file, start by counting the frequencies of each character and assign the shortest code to the character that appears the most frequently.
Then encode the file by building a binary tree where you sum the frequency of each character from bottom-up.
Assume we have a text file where Z and K appear the least amount of times. We start by adding their frequencies ($2 + 7 = 9$) and add a parent node with that summed frequency. Continue along the list until the tree is built.
When confronted with a string of bits, we simply traverse the tree in the order dictated by the edges in the tree. For example, consider the string 11100 for the above tree.



In this case, we would start at the node, then go right three times, then go left and land on the root node for the character C. Thus 1110 would encode the character C, and we move on to the next part of the string, which is just a 0, and that simply encodes E. Therefore 11100 would encode CE.
Since we need to store the binary tree, Huffman coding does not make sense for small text files.

1.2.1 How to construct a Huffman tree step-by-step

Step 1:

- Create a leaf node for each unique character.
- Build a min heap using these leaf nodes.
  - Min Heap is used as a priority queue.
  - The value of the frequency field is used to compare two nodes in the min heap.
  - Initially, the **least** frequent character is at root.

Step 2:

- Extract two nodes with the minimum frequency from the min heap.

Step 3:

- Create a new internal node with the sum of the frequencies of the two nodes selected in step 2.
- Now give this new internal node two children.
  - The left child is the first extracted node from step 2.
  - The right child is the second extracted node from step 2.
- Add these nodes to the min heap.

Step 4:

- Repeat steps 2 and 3 until the tree is done.

1.3 Prim, Kruskal and Dijkstra comparison

Finding shortest paths from a single source vertex and finding a minimum spanning tree are graph optimization problems solvable by greedy algorithms. We looked at Prim's algorithm in the lecture. Consider Kruskal's and Dijkstra's algorithms. For each of these algorithms, describe:

- what is the greedy choice made in each step of the algorithm?
- what is the sub-problem that remains after each step of the algorithm? (Remember, the sub-problem should "look" like the original problem, only smaller)

In **Prim's algorithm**, you keep track of a "visited" list, and you always select the edge with the lowest cost that connects a visited node to an unvisited node. In each step of each algorithm, the sub-problem that remains is to pick the shortest path (or lowest cost edge) that leads to an unvisited node.
In **Kruskal's algorithm**, you always select the edge with the lowest cost if one or both of the nodes is unvisited. In each step of each algorithm, the sub-problem that remains is to pick the shortest path (or lowest cost edge) that connects to one or more unvisited nodes.
In **Dijkstra's algorithm**, you always visit the cheapest, unvisited node by picking the edge with the lowest cost. In each step of each algorithm, the sub-problem that remains is to pick the shortest path (or lowest cost edge) that leads to an unvisited node (same as Prim's).

2 All-pairs shortest path

When working with All-pairs shortest path algorithms, typically you use different matrices to keep track of data.
An algorithm to find the shortest path between all vertices in a graph cannot use a greedy approach. We have to find every possible solution, and thus we use dynamic programming instead.

2.1 Floyd Warshall

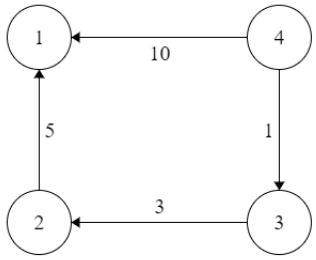
The most typical algorithm for all-pairs shortest paths problems is the **Floyd Warshall** algorithm, however you can also use Dijkstra's shortest path algorithm for graphs without negative-cost edges. It's worth noting that Dijkstra only finds the shortest path between two vertices, so you would have to run it for every possible pair of vertices to find the all-pairs shortest paths.
Floyd Warshall finds the shortest path for every single vertex pair in $\mathcal{O}(n^3)$ time, the same as Dijkstra when used to find for all-pairs shortest path.
As an **input** to all-pairs shortest paths algorithms, you use an *adjacency matrix*, which is an N x N matrix where each entry tells us something about the edges E in the graph:

- If i = j, the entry is 0.
- If (i, j) are in E, the entry contains the weight of the directed edge (i, j)
- If (i, j) are *not* in E, the entry is ∞.

Here's an example of an adjacency matrix for graph G:

G	j = 1	j = 2	j = 3	j = 4
i = 1	0	5	∞	10
i = 2	∞	0	3	∞
i = 3	∞	∞	0	1
i = 4	∞	∞	∞	0

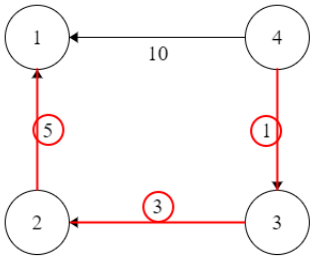
Notice how the diagonal are all zeroes as we defined above since i = j. Using the adjacency matrix above, we can construct the DAG which would look something like this:



The **output** of all-pairs shortest paths algorithms is a *distance matrix* and a *predecessor matrix*.
The distance matrix is an N x N matrix where each entry contains the weight of a shortest path from vertex i to j. Once more, the entry is infinite if no path from vertex i to j exists. The distance matrix for the above graph would be:

dist(G)	j = 1	j = 2	j = 3	j = 4
i = 1	0	5	8	9
i = 2	∞	0	3	4
i = 3	∞	∞	0	1
i = 4	∞	∞	∞	0

Notice how for i = 4 and j = 1, we now see that the weight of the shortest path is 9, as shown below:



The predecessor matrix is an N x N matrix where each entry contains the predecessor of j on a shortest path from i. If i = j or if there is no path from vertex i to vertex j, the entry is Nil.
For the above table, the predecessor matrix is:

pred(G)	j = 1	j = 2	j = 3	j = 4
i = 1	NIL	2	2	2
i = 2	NIL	NIL	3	3
i = 3	NIL	NIL	NIL	4
i = 4	NIL	NIL	NIL	NIL

Using this, we can easily compute the **transitive closure** between two vertices by simply using resulting matrices. In other words, a biproduct of the algorithm is being able to easily tell which vertices can be reached from any vertex using the output matrices. If the shortest path from i to j remains ∞ after running Floyd's algorithm, you can be sure that no directed path exists from i to j.

Checking what adjacency matrix matches a distance matrix - Check if any of the listed adjacency matrices have a lower value than the given distance matrix. If so, then that matrix cannot be the correct answer, because the distance matrix entries can only be less than or equal to the entries in the adjacency matrix

2.1.1 How to do the Floyd-Warshall algorithm

Use the following formula:
Iterate from $i = 1$ to $j = 1..N$

$$dist[i, j] = min(dist[i, j], dist[i, k] + dist[k, j])$$

Repeat up to $i = N$.

An example can be seen below:

$i = 2$, iterate over $j = 1$ to 6:

$$dist[2,1] + dist[1,1] = 1 + 0 = 1$$
$$dist[2,1] + dist[1,2] = 1 + \infty = \infty$$
$$dist[2,1] + dist[1,3] = 1 + \infty = \infty$$
$$dist[2,1] + dist[1,4] = 1 + \infty = \infty$$
$$dist[2,1] + dist[1,5] = 1 + -1 = 0$$
$$dist[2,1] + dist[1,6] = 1 + \infty = \infty$$

Each i iteration corresponds to updating the i'th row of the distance table. Only update the cell if the new number is smaller than the current number in the cell.
The new entries in the predecessor table correspond to the k'th iteration. If you have $k = 2$, all the entries in the predecessor, for that iteration, must be 2.

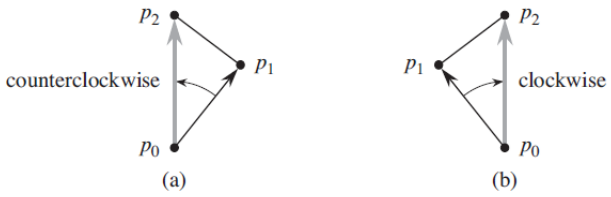
3 Computational geometry

3.1 Line sweep algorithm

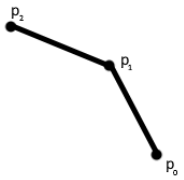
3.1.1 The cross product

In most computation geometry algorithms, it is often useful to know if two two key properties about line segments segments:

- Given two directed line segments $\overrightarrow{p_0, p_1}$ and $\overrightarrow{p_0, p_2}$, is $\overrightarrow{p_0, p_1}$ clockwise from $\overrightarrow{p_0, p_2}$?
- Given two directed line segments $\overrightarrow{p_0, p_1}$ and $\overrightarrow{p_1, p_2}$ do we make a left turn at point p_1 if we traverse $\overrightarrow{p_0, p_1}$ and then $\overrightarrow{p_1, p_2}$?



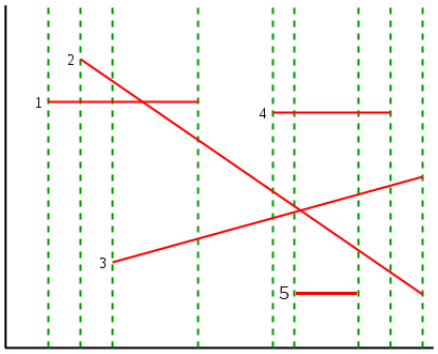
This can be calculated using the cross product defined by: $(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0) \cdot (y_2 - y_0) - (x_2 - x_0) \cdot (y_1 - y_0)$
When this cross product is positive, $\overrightarrow{p_0, p_1}$ is clockwise from $\overrightarrow{p_0, p_2}$, and counter-clockwise if negative.



For two consecutive line-segments $\overrightarrow{p_0, p_1}$ and $\overrightarrow{p_1, p_2}$, if the sign of the cross product is negative, we make a left turn at p_1 - and vice versa - which can be seen above. A cross product of 0 means that the points p_0, p_1 and p_2 are co-linear, meaning their directions are identical or opposite.
We use the cross product to answer these questions instead of computing the angle because it is faster and more precise. When segments are nearly parallel, finding angles is too sensitive to the precision of the divison operations on computers.

3.1.2 Line sweep algorithm for line intersections

The line sweep algorithm is used to figure out which lines in a set intersect. It has many applications, especially in computer graphics, like building a voronoi diagram.
The algorithm requires n lines and $2n$ points such that each line consists of two end points.



Sweep Lines

First, sort all the line end points from left to right based on their x coordinate such that the left-most points are first in the list, and the right-most points are last. Break ties by putting points with lower y-coordinates first. While sorting, maintain a flag that indicates whether the point is the left-most or right-most point of the line.
Start from the left-most point.
For each point:

- If the current point is a *left point* of its line segment:

  - Add its line to the active line segments.
  - The active line segments are line segments where we have seen the left-most end point, but we have not yet seen the right-most point.
  - Check if its line segments intersects with the **active** line segments directly above and below (if they exist).

- If the current point is a *right point* of its line segment:
  - Remove the line segment from the active list.
  - Check whether the active line segments directly above and below intersect with each other.

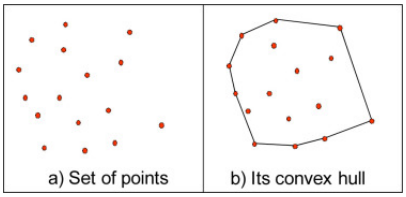
This procedure mimmicks passing a vertical line through each point from left to right, hence the name **line sweep**. Each point is known as an **event point**. In implementing this algorithm, we need an efficient way to store all active line segments. We need to be able to do the following operations efficiently:

- **Insert** a new line segment.
- **Delete** a line segment.
- Find the line segment immediately **above** a segment.
- Find the line segment immediately **below** a segment.

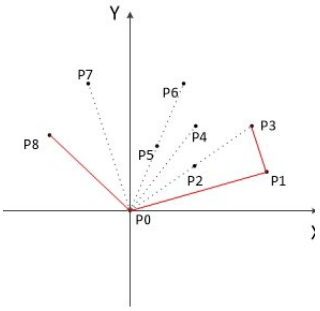
Therefore, we use a Self-Balancing Binary Search Tree (BST) data structure to keep track of the active line segments (usually a Red Black Tree). Using this, we can do all the above operations in $\mathcal{O}(log(n))$ time. BSTs provide an efficient implementation of **mutable, ordered lists**. This is useful in the line sweep algorithm since we often want to find the active neighbors of a given line segment.

3.2 Convex hull - Graham Scan and Jarvis March

In geometry, the *convex hull* of a shape is the smallest convex set that contains it.



In Graham Scan, the bottom-most point is first found. Then all the other points are sorted based on their counter-clockwise angle with respect to the bottom-most point, as can be seen below.



We then create a stack initially containing the bottom-most point and the first point in the sorted list. Then we iterate the sorted list of points, placing each point on the stack, but only if it makes a **counter-clockwise** turn relative to the previous two points on the stack. Otherwise, if it makes a clockwise turn, the previous point is popped off the stack. Continue doing this until the convex hull is generated.
Jarvis's March also starts by finding the bottom-most point, but instead of sorting, it loops through all the points again in a brute-force manner to find the point that makes the smallest counter-clockwise turn with respect to the previous point. It repeats this process until it reaches the starting point.

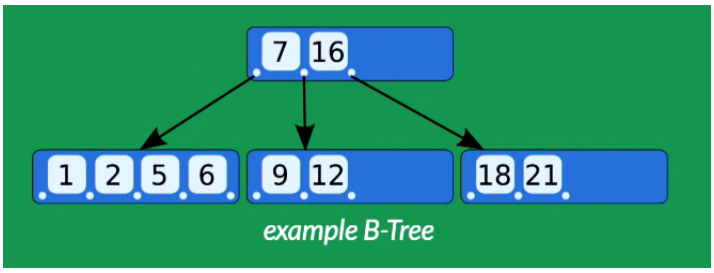
3.2.1 Runtime complexity and comparison

Jarvis March is $\mathcal{O}(n \cdot h)$ where h is the number of vertices, meaning that if $h = n$ then the worstcase runtime is actually $\mathcal{O}(n^2)$. Graham Scan has time complexity $\mathcal{O}(n \cdot log(n))$. When $h \leq log(n)$, Jarvis's March is faster than Graham's scan; otherwise, it is slower.
Here is a video explanation of the two algorithms: `https://www.youtube.com/watch?v=B2AJoQSZf4M`

4 External-Memory Algorithms and Data Structures

4.1 B-Trees

A B-tree is a data structure where the nodes may contain $n-1$ number of values and n number of keys/pointers to child nodes. The tree is self-balancing and the nodes are sorted such that each value indicates which pointer to follow when searching for data in the tree.
The following is an example of a B-tree:



As can be seen here, each value in the node to the left is lower than 7, and each value in the node following the pointer to the right of 7 is greater than 7.
If the lead node is full, split the node in two, with the smaller half of the items in one node and the larger half in the other. "Promote" the median item to the parent node. If the parent node is now full, split it and repeat. If this reaches the root node, the height of the tree will grow by one.
The key difference between a B tree and a binary search tree is that the tree is self balancing, and so all operations on it take only $O(log(n))$ whereas operations on an unbalanced binary search tree take time $O(n)$.
B trees are useful when working with memory because they allow you to more efficiently search for and insert data on external memory without having to perform slow disk operations every time. For example, if you cannot have all your data loaded into main memory (as is usually the case), having just the root node loaded into main memory and using its keys to access data is both much faster and less memory consuming.

4.2 B$^+$-Trees

B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in a B+ tree, records (data) can *only* be stored on the leaf nodes while internal nodes can only store the key values.
The **leaf nodes** of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

4.2.1 How to build B$^+$-Trees

Things to keep in mind:

- Minimum fanout is the minimum number of elements that can be stored in each leaf node.
- Maximum fanout is the maximum number of elements that can be stored in the leaf nodes before they have to be split.
  - If the maximum fanout is 4, you split when the 5'th element is inserted.

- When splitting the leaf node, the middle element is made an internal node, e.g. it is pushed up and linked to the two leaf nodes.

  If the amount of elements in the leaf node is *even*, you pick the **leftmost** of the middle elements.

Step 1 - Inserting

- Insert each element of the input into the tree.
- When inserting make sure to insert the elements so that they are in order i.e. alphabetical etc.

Step 2 - Splitting

- When maximum fanout is achieved, you split the node.
- When splitting the node, the middle (or leftmost of the middle) elements is pushed up to become an internal node.
- Update the pointers between the new internal node to the leaf nodes. **There should not be pointers from internal nodes to other internal nodes**.

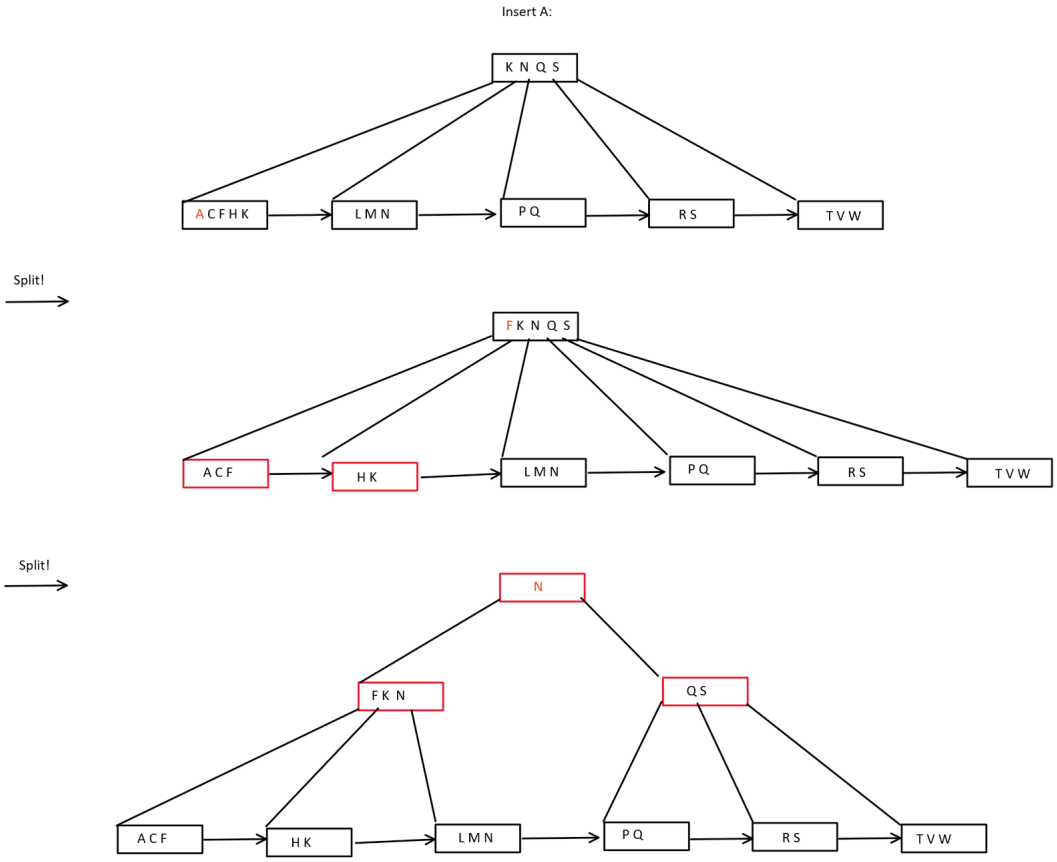


Figure 1: Example of how to split a B^+-tree

4.3 External memory merge sort

N = number of data elements (records)
B = number of data elements a disk can store
m = available main memory pages
n = the number of disk pages in the initial file

First phase iterations for external memory sort

$$Runs = \frac{Disk\ pages}{Pages}$$

Second phase iterations for external memory sort

$$\lceil log_{m-1}(Runs) \rceil$$

Number of IO Operations performed

$$IO - OPs = (\#\ of\ 2nd\ phase\ iterations\ + 1) \cdot 2 \cdot Disk\ pages$$

5 Multithreaded algorithms

- **Strand:** One or more instructions grouped such that the group contains no parallel control (no spawn, sync, or return from a spawn - via either an explicit return statement or the return that happens implicitly upon reaching the end of a procedure).
- **Critical path:** The longest path through the dag, corresponding to the longest time to perform any sequence of jobs.
- **Work:** The running time on a machine with one processor (T_1). The number of vertices.
- **Span:** The running time on a machine with infinite processors (T_∞). The length of the critical path.
- **Parallelism**: $\frac{work}{span}$ - can be thought of as the gain we get from threading/making something parallel.

In the example below, the critical path contains 8 strands numbered in red.
Useful symbols and formulae:

- P = Number of computation units.
- T_1 = Work, running time on a machine with 1 processor.
- T_∞ = Span, running time on machine with infinite processors.
- T_P = Running time on machine with P processors.
- $T_P \geq T_1/P$ = Work law.
- $T_P \geq T_\infty$ = Span law.
- T_1/T_P = Speedup, speedup must be $\leq$ P and if it's equal to P we achieve a perfect speed up.
- T_1/T_∞ = Parallelism, maximum number of speedup that can be achieved.
- $T_1/PT_\infty = slackness = \frac{Parallelism}{P}$. If $slackness < 1$ we can never achieve a perfect speed up.

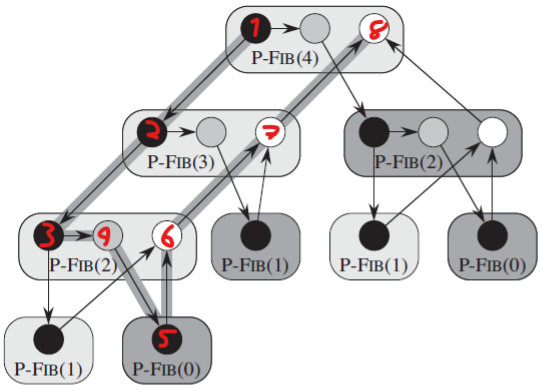


Figure 27.2 A directed acyclic graph representing the computation of P-FIB(4). Each circle represents one strand, with black circles representing either base cases or the part of the procedure (instance) up to the spawn of P-FIB($n-1$) in line 3, shaded circles representing the part of the procedure that calls P-FIB($n-2$) in line 4 up to the **sync** in line 5, where it suspends until the spawn of P-FIB($n-1$) returns, and white circles representing the part of the procedure after the **sync** where it sums x and y up to the point where it returns the result. Each group of strands belonging to the same procedure is surrounded by a rounded rectangle, lightly shaded for spawned procedures and heavily shaded for called procedures. Spawn edges and call edges point downward, continuation edges point horizontally to the right, and return edges point upward. Assuming that each strand takes unit time, the work equals 17 time units, since there are 17 strands, and the span is 8 time units, since the critical path—shown with shaded edges—contains 8 strands.

6 Turing Machines

DEFINITION 3.3

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

- A Turing Machine can write the blank symbol on its tape, since $\sqcup$ is part of the tape alphabet according to the definition.

- The tape alphabet **cannot** be equal to the input alphabet, however it will contain all the symbols of the input.
- A Turing Machine *can* stay on the same cell for two subsequent steps of a computation *if* it is the leftmost cell on the tape, and you take another left step—in that case, you stay on the cell.
- A Turing Machine *must* contain both an accept and reject state.
- Turing machines are deterministic and the machine can only be in one distinct state after a transition.

6.1 Languages for Turing Machines

Some general things to note about languages for Turing Machines:

- A language is not an input for a Turing machine (TM) and we cannot run a TM on a language and ask if it halts or not. We can only ask whether a TM halts on a given string.
- A TM cannot be a member of a language.
- Decidable languages are subsets of recognizable languages, and therefore a decidable language is also recognizable.

6.1.1 Definition of Turing-recognizable

A language is "Turing-Recognizable" iff there exists a Turing Machine such that

- when encountering a string in that language, the machine terminates and accepts that string;
- when encountering a string not in that language, the machine either terminates and rejects that string or doesn't terminate at all.

6.1.2 Definition of Turing co-recognizable

A language L is Turing co-recognizable iff its complement is recognizable —in other words, iff there exists a verifier for $\bar{L}$.
Thus, to see if a language is co-recognizable, we can ask: Given a string $w \notin L$, could you prove that $w \notin L$?
Basically, if you give the Turing Machine a string **no**t in the language, it will eventually confirm that it is **not** in the language.

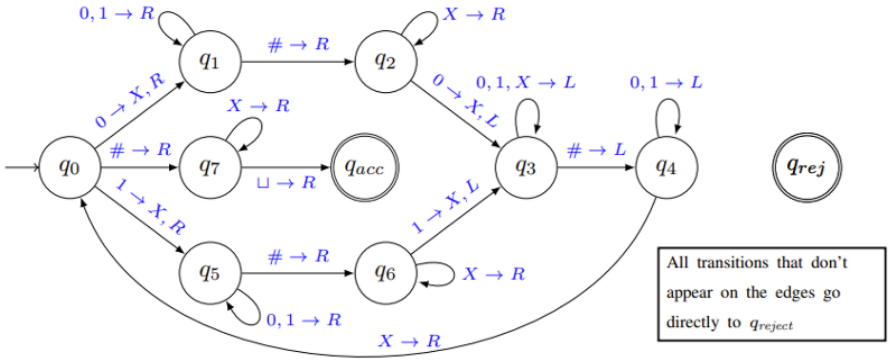
6.1.3 Definition of decidability

A language $L \subseteq \Sigma^*$ is decidable iff there exists a Turing Machine that accepts the string when the input is a member of the language, and rejects the string when the input does not lie in the language.
A language is decidable if it is both **Turing-recognizable** and **Turing-co-recognizable**.

6.1.4 How to write the sequence of configurations for a TM given an input string

When you are given an exercise where you have to give the configurations of a TM given some input string, you should understand the following:

- The given input string will usually look something like: 10#10
- This corresponds to the transitions that you have to take in the state diagram. An example of such a state diagram can be seen below.
- Each transition will be on the form: y → x, R (or) L
  - y is the symbol you are reading off the input.
  - x is what you write (you replace the original value). Sometimes x is not included, in which case nothing is written to the tape.
  - R and L correspond to which direction you move the head (right and left).



An example could be the sequence of configurations for the input 10#10. Using the picture, one can see that for the first step in the sequence the configuration will transform like the following:

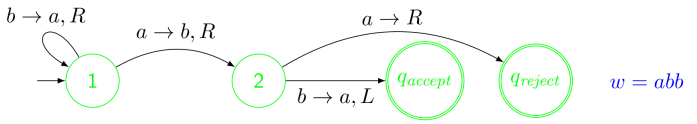
- Starting in the initial state q_0, you have the initial configuration $q_0$10#10.
- You read a 1.
- You write an X to the tape, which replaces the 1 with an X.
- You move the head to the right and go to state q_5.
- The configuration will look like X$q_5$0#10.

6.1.5 Representing Turing Machines as a string

Consider:

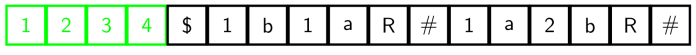
- A Turing Machine takes as input an arbitrary string w.
- A Turing Machine can be encoded as a string.

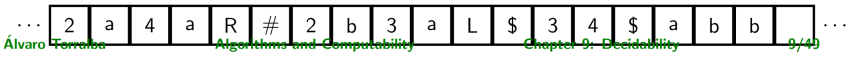
So: **We can pass the description of a Turing Machine M to another Turing Machine M'.** The following images show an example of how a specific Turing Machine is represented as a string, starting with the states $(1, 2, 3, 4)$, then the transitions, the accept and reject states, and finally the input string w. Notice how \$ is used as a separator between each Turing Machine "category" (states, transitions, accept/reject and the input string w), and # is used as a separator between transitions.

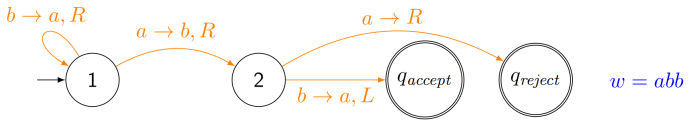


→This is commonly done with compilers, or virtual machines

1. States (initial state on the 1st place)

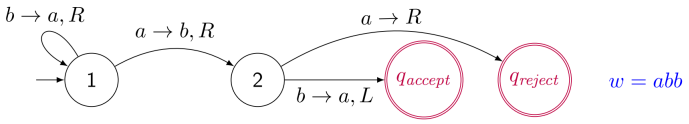






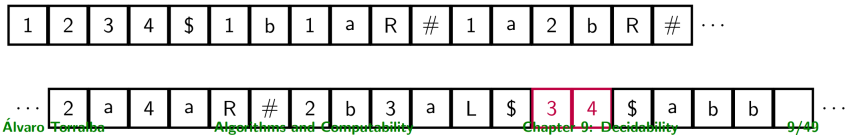
→This is commonly done with compilers, or virtual machines

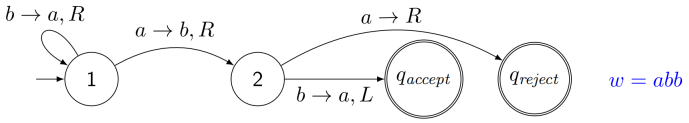
1. States (initial state on the 1st place) 2. Transitions



→This is commonly done with compilers, or virtual machines

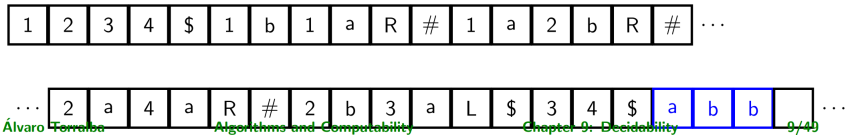
1. States (initial state on the 1st place)
2. Transitions
3. Accept/Reject states





→This is commonly done with compilers, or virtual machines

1. States (initial state on the 1st place)
2. Transitions
3. Accept/Reject states
4. String (w)



---

7 P = NP

7.0.1 Common Big-O functions

- $\mathcal{O}(1)$ - constant time
- $\mathcal{O}(log(n))$ - logarithmic time
- $\mathcal{O}((log(n))^c)$ - polylogarithmic time
- $\mathcal{O}(n)$ - linear time
- $\mathcal{O}(n^2)$ - quadratic time
- $\mathcal{O}(n^k)$ - polynomial time
- $\mathcal{O}(k^n)$ - exponential time
- $\mathcal{O}(n!)$ - factorial time

7.0.2 The two definitions for NP

The first definition for NP is the standard definition and it states:

$$NP = \bigcup_{k \in \mathbb{N}} NTIME(n^k)$$

Where $NTIME(n^k)$ is the set of decision problems can be solved by a nondeterministic Turing machine in $\mathcal{O}(n^k)$ (polynomial) time.

The second definition for NP is using a polynomial time verifiers. This states:
A verifier for a language L is a decider V such that:
$L = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$

- The string c is called a certificate or a proof.
- A verifier is called a polynomial time verifier if it runs in a polynomial time in the length of w (hence the length of c is irrelevant).
- A language L is a polynomial time verifiable language if it has a polynomial time verifier.

In simple terms, the certificate will be the variable assignment, and it can be checked whether such a certificate is correct in polynomial time. If it can be checked in polynomial time, the language is in NP.

7.0.3 Definitions for the class P and languages that belong to P

Below are some definitions for the class P:

- P is the class of all languages that are decidable by deterministic single-tape Turing machines running in polynomial time.
- A language L belongs to P iff there is a constant k and a decider M running in time $\mathcal{O}(n^k)$ such that L = L(M).

Below are 5 languages that belong to P:

- PATH
- Ø
- $\{a^k b^k \mid k \geq 0\}$
- $\{\langle G \rangle \mid G \text{ is a connected graph}\}$
- $\{\langle M \rangle \mid M \text{ is a TM which has more than 10 states}\}$

7.0.4 Definition of Polynomial Time Mapping Reducibility

Let $A, B \subseteq \Sigma^*$.
We say that language A is **polynomial time (mapping) reducible** to language B, written $A \leq_P B$, iff there is a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ such that, for every $w \in \Sigma^*$,

$$w \in A \text{ if and only if } f(w) \in B$$

7.0.5 Common problems for mapping reductions

- $REGULAR_{TM}$ test given a TM M if L(M) is regular

  $$REGULARTM = \{\langle M \rangle\} | M \text{ is a TM s.t. } L(M) \text{ is regular}$$

- A_{TM} is a TM that test if given a TM M and a string w, does the Turing machine accept w?

  $$\text{ATM} = \{\langle M, w \rangle\} | M \text{ is a TM} w \text{ a string, and } M \text{ accepts } w$$

- $HAMPATH$ is a path that visits each vertex exactly once,

  Hampath = $\{\langle G, s, t \rangle\}|$ G is a digraph with a hamiltonian path from s to t

- SAT A Boolean formula ϕ is satisfiable if there is an assignment of truth values to the variables on which ϕ evaluates to true

  $$\text{SAT} = \{\langle \phi \rangle\} | \phi \text{ is a satistiable Boolean formula}$$

- E_{TM} test given a TM M if the language of M empty

  $$\text{ETM} = \{\langle M \rangle\} | M \text{ is a TM s.t. } L(M) = \emptyset$$

- EQ_{TM} test given two TMs M_1 and M_2 if they recognize the same language

  $$\text{EQTM} = \{\langle M_1, M_2 \rangle\} | M_1 \text{ and } M_2 \text{ are TMs s.t. } L(M_1) = L(M_2)$$

REGULARTM = $\langle$M$\rangle$ — M is a TM s.t. L(M) is regular

7.0.6 Proving that a problem is NP-complete

1. A problem is NP-hard if every problem in NP polynomially reduces to it, here we assume that every problem in NP-hard is as least as hard as NP however not all NP-hard problems belong to NP
2. A problem is NP-complete if it is both in NP and NP-hard
3. First, prove that the problem is in NP using the certificate method

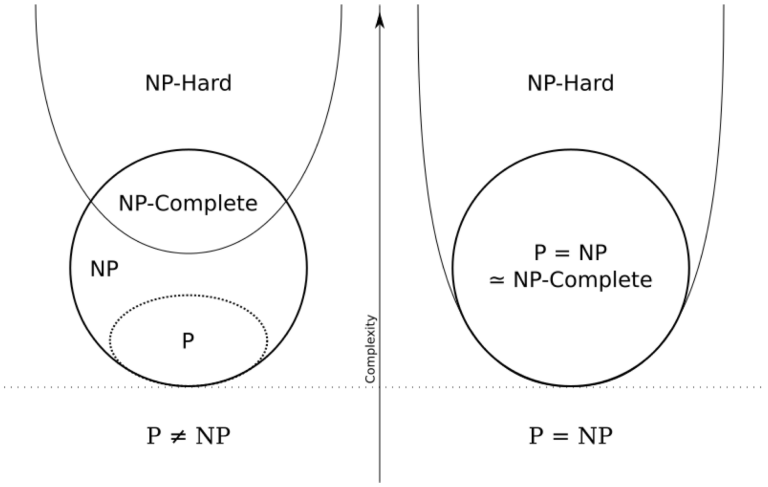
- If we can prove the certificate in polynomial time then it's NP

4. We then prove that it is NP-hard by choosing another NP-hard problem and reducing it to our current problem

T(M) belongs to which classes The points below assumes that we don't know if P = NP or P != NP

- May: Includes all the classes a Turing machine could belong to with the given run-time
  - In case of the run-time is $\mathcal{O}(number^n)$ there is the possability of $n = 1$ which means the run-time could be polynomial.
- Must: Includes all the classes a Turing machine **Must** belong to with the given run-time

In case we assume that we know if P = NP or P!=NP then it changes whether all problems in P are also in NP-Complete.



8 Multitape Turing Machines

A multi-tape Turing machine is a variant of the Turing machine that utilizes several tapes. Each tape has its own head for reading and writing. Initially, the input appears on tape 1, and the others start out blank.
Note that some proofs require you to think about this. In such proofs you just copy the input to the second (or other) tape(s).

This model intuitively seems much more powerful than the single-tape model, but any multi-tape machine—no matter how many tapes—can be simulated by

a single-tape machine using only *quadratically* more computation time.

Thus, multi-tape machines *cannot* calculate any more functions than single-tape machines, and none of the robust complexity classes (such as polynomial time) are affected by a change between single-tape and multi-tape machines.

Multitape Turing machines can be used to simulate nondeterminism.

8.0.1 Complexity of multitape to singletape Turing machines

How to calculate the complexity of a single-tape Turing Machine corresponding to a multitape Turing Machine:

This is straight forward. If the complexity of the multitape Turing machine is given as $\mathcal{O}(n^3)$, all you have to do is square the complexity:

$\mathcal{O}(n^3)^2 = \mathcal{O}(n^6)$

This is due to the fact that when converting to a single tape, the machine will have to go forward once to find the corresponding places to update and backwards once on the tape to update said values.

9 Cook-Levin legal windows

Each window represents a section of the configuration of a nondeterministic Turing machine. The first row of the window represents the given section of configuration at time T and the second row of the window represents the section of configuration at time $T + 1$.
The transition functions given in the exercise will be of the form: δ(q, x) = {q, y, R/L} where the left side of the equation is "You are in state q and read x" and the right side of the equation means "you transition to q, write a y and move either right or left".
Note that there may be several sets of options for what happens when you transition.
Assuming the head is currently in state q, you could consider several options when reading the cell.

- If q is not present in the first row, then q can be anywhere. You therefore decide where q may be and where the following transition occurs from.
- If a # symbol is present in the first row on the far left, it signifies that you are in the beginning of the configuration. You therefore cannot move further to the left and the cell just below should also be a #.
- If a # is present in the first row on far right, it signifies that you are at the end of the configuration. You therefore cannot move further to the right and the cell just below should also be a #.

It should also be noted that q is not part of the string. q is the location of the head. Therefore if you see for example: aaq that means there is technically a symbol to the right of the q that is not shown in the window. Similarly if you see: aqa it means that you are in state q and reading an a. You could think of q and the following symbol as a unit.
Below is an example of an exercise using this. We have also drawn the states and transitions as an automata to help visualize the transition functions.

Consider a nondeterministic Turing machine M over the input alphabet $\Sigma = \{a\}$, tape alphabet $\Gamma = \{a, \sqcup\}$, control states $Q = \{q, q_{accept}, q_{reject}\}$ and the following δ function:

- $\delta(q, a) = \{(q, a, R), (q_{reject}, \sqcup, L)\}$
- $\delta(q, \sqcup) = \{(q_{accept}, a, R)\}$

Fill in all legal windows (recall the proof of Cook-Levin Theorem) with the following chosen first rows. Note that this covers only some of the legal windows. The number of windows in every row indicates how many possible legal windows you should be able to find. You might use the abbreviations q_a for q_{accept} and q_r for q_{reject}.

a	a	a

a	a	a

a	a	a

a	a	a

#	a	a

#	a	a

a	q	a

a	q	a

a	q	⊔

q	a	a

q	a	a

q	a	a

q	a	a

a	a	q

a	a	q