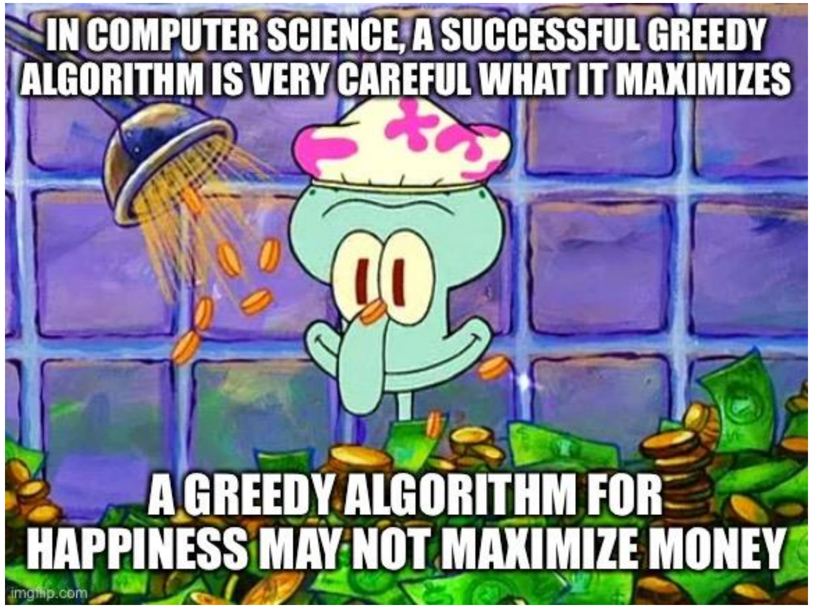
D4: Greedy Algorithms and Introduction to Computability



Sec 101: MW 3:00-4:00pm DOW 1018
IA: Eric Khiu

Announcement

- Homework 2 due Thursday May 23
- Homework 3 due Tuesday May 28
  - Will be released latest by tomorrow
- Midterm
  - Midterm: May 30 7-9pm
  - Coverage: Up to Lecture 12: Reducibility
  - Past midterms are posted
  - Midterm review session: May 28 DOW 1013 6-8pm
  - Midterm format will be announced sometime this week
  - Midterm room assignment will be announced by today

Regarding terminologies

In the context of EECS 376

- **Maximal/ minimal** solution: usually refer to local max/ min
- Use **maximum/ minimum** to indicate global max/ min for clarity
- We don't say an algorithm is optimal (adj.)
- We say the *output*/ solution given by an algorithm is optimal

- Spoiler alert: Today we will be learning a lot of new vocab

Agenda

- Greedy Algorithm
- Intro to Computability
- DFA

Greedy Algorithms

Notes

Greedy Algorithms

- **Intuition:** Take the local “optimal action” at every step
- Starter: What is the smallest number of coins that sum to 30¢?
  - 30¢: 1 quarter + 1 nickel = 2 coins
- What is the general strategy?
  - Always picks quarter if possible
  - Pick dimes if possible
  - Pick nickels if possible
  - Pick pennies if possible
- What is the local “optimal action” here?
  - Pick coins with highest denomination
- Does greedy always work?

Penny		1¢
Nickel		5¢
Dime		10¢
Quarter		25¢

Greedy Algorithms

- Suppose we remove nickel from the money system, what is the number of coins given by the greedy approach for 30¢?
  - 1 quarter + 5 penny = 6 coins
  - But we can do this using 3 dimes = 3 coins!
- Takeaway: Greedy is not always optimal
  - It’s often not optimal
  - But: Can be useful for heuristics

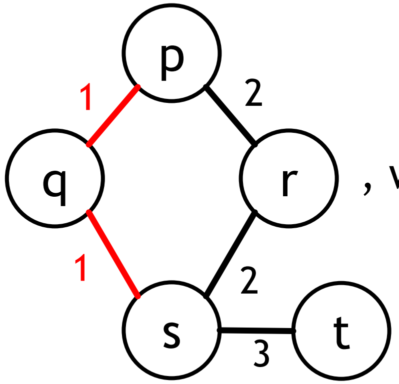
- Always picks quarter if possible
- Pick dimes if possible
- ~~Pick nickels if possible~~
- Pick pennies if possible

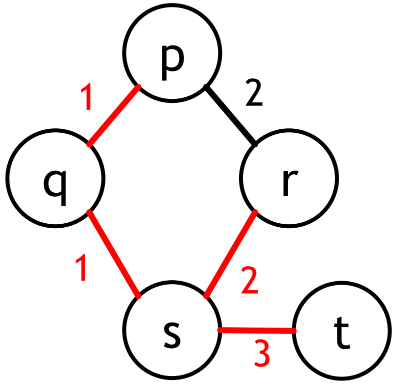
Penny		**1¢**
~~**Nickel**~~		~~**5¢**~~
Dime		**10¢**
Quarter		**25¢**

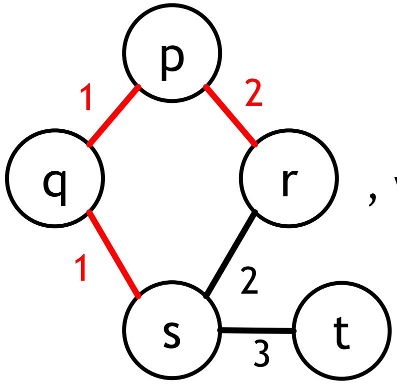
Induction with Exchange Argument

- Exchange argument
  - Show that we can transform any optimal solution into the solution given by our algorithm by exchanging each piece of it out one-by-one without increasing the final cost.
- Induction framing
  - **Base case:** simplest form of the problem – often zero
  - **Inductive Hypothesis:** Assume that the first k choices of the greedy solution are part of *some* optimal solution
  - **Inductive step:** Show that the first $k+1$ choices (not just the $(k+1)^{th}$!) are also part of *some* optimal solution

Induction with Exchange Argument

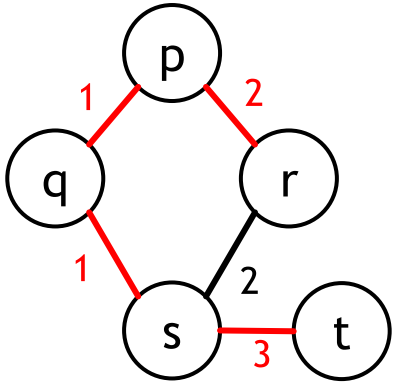
- Example: Running Kruskal for MST
  - Suppose on the k^{th} iteration gives 

, which is part of 

OPT
  - The, the $(k+1)^{th}$ iteration gives 

, which is **not** part of OPT!

Discuss: Why is it ok to exchange (s,r) for (p,r)?
A. Because both actions add r to the current tree
B. Because (s,r) and (p,r) have the same weight
C. Because it is still part of *some* MST



Greedy Line Copying

- You are copying a list of n problems from your textbook to the paper you do your homework on. You need to copy all the problems over, and you want to have them in order. Formally, the ith problem is l_i lines long, and you're copying them onto an ordered set of sheets of paper which are each m lines long
  - The problems must be copied in the assigned order
  - All problems must be copied
  - Problems must be kept on one sheet of paper and can't be split across them.

- Describe a greedy algorithm to **minimize** the number of pages you need

Greedy Line Copying

- Describe a greedy algorithm to minimize the number of pages you need
  - Put as many problems possible onto the first page until the next problem would exceed the m limit
  - Then put the next problem onto the next page, until all n problems are copied

Sanity check: How do know if the next problem would exceed the m limit?

Greedy Line Copying

GreedyLineCopying:

- Put as many problems possible onto the first page until the next problem would exceed the m limit
- Then put the next problem onto the next page, until all n problems are copied

- Prove that the algorithm produces optimal result using exchange argument
- Reminder:
  - **Inductive Hypothesis:** Assume that the first k choices of the greedy solution are part of *some* optimal solution
  - **Inductive step:** Show that the first $k+1$ choices (not just the $(k+1)^{th}$!) are also part of *some* optimal solution
- Outline
  - Let G denote the solution given by our greedy algorithm
  - P(k) = The first k $\boxed{}$
  - Base case: $\boxed{}$
  - IH: There exists $\boxed{}$
  - Goal: Show that $\boxed{}$

Greedy Line Copying

GreedyLineCopying:

- Put as many problems possible onto the first page until the next problem would exceed the m limit
- Then put the next problem onto the next page, until all n problems are copied

- Outline
  - Let G denote the solution given by our greedy algorithm
  - P(k) = The first *k* pages are the same as *some* optimum solution
  - Base case: *k*=0: Nothing to prove
  - IH: There exists some optimal solution OPT whose first k pages are the same as those in G
  - Goal: Find an optimal solution OPT' whose first k+1 pages are the same as those of G

Greedy Line Copying

GreedyLineCopying:

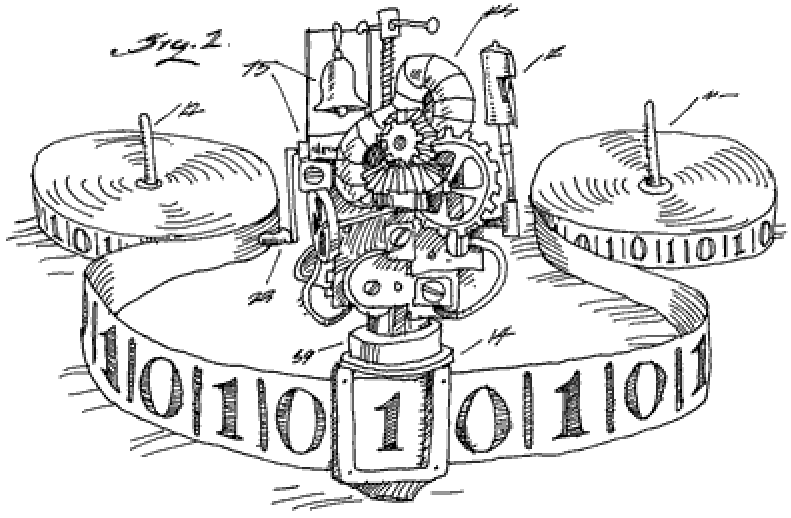
- Put as many problems possible onto the first page until the next problem would exceed the m limit
- Then put the next problem onto the next page, until all n problems are copied

- Prove that the algorithm produces optimal result using exchange argument
  - Inductive step: By IH the first k pages are identical, now consider the (k+1)st step
  - Case 1: (k+1)st are equal: Done
  - Case 2: (k+1)st are not equal: Consider the problems that are on the (k+1)st page of G but not on the (k+1)st page of OPT, where should they go in OPT?
    - The start of (k+2)nd because the problems must be copied in order
  - Construct OPT’ by copying those problems onto the (k+1)st page until it matches the (k+1)st page of G. This will not increase the number of pages use. Why?
    - Because we are only moving problems forward

TL; DPA

- We talked about the greedy algorithms that computes a solution to an optimization problem by making (and committing to) a sequence of locally optimal choices.
- Greedy algorithms do not always produce a global optimum solution. In that case, we give a counterexample to disprove its optimality.
- For some specific problems and greedy algorithms, we can prove that the result is indeed a global optimum using the exchange argument.

Unit 2: Computability



Sec 101: MW 3:00-4:00pm DOW 1018
IA: Eric Khiu

Introduction to Computability

Notes

Intro to Computability

- Motivation: we want to know "what problems can a computer compute", or more interestingly, "what problems *can't* a computer compute"?

- In this unit,
  - We will first structure what "problem" means
  - Then, we will discuss what "computer" means

Decision Problems

- Problems that have a 'yes' or 'no' answer
- Examples:
  - Can we fit k apples into a paper bag?
  - Given two integers x and y, is $x = y$?
  - Is it possible to get from A to B in under z miles of travel?
- Parametrize "Is it possible to get from A to B in under z miles of travel?"
  - Define $w = (A, B, z)$, where each w has a yes or no answer
  - The parametrization tells us how the input look like

Languages

- The language of a decision problem is the set of all 'yes' inputs to that problem

$$L_{canTravel} = \{(A, B, z) : \text{ There is a path to travel from A to B in less than z miles}\}$$

- We say $w = (A, B, z) \in L_{canTravel}$ if and only if there is a path from A to B in less than z miles

- A language is a set, so all set operations are valid, for instance:

$$\overline{L_{canTravel}} = \{(A, B, z) : \text{ There is no way to travel from A to B in less than z miles}\}$$

Wait! Computers can't read your input!

- To make an input computer readable, we need to encode it as a string
- But it can't be *any* string, we specify the alphabet (Σ) – a finite set of characters allowed to be used in the string
  - Example: {0, 1}, {1, …, 9}, {a, b, c}, set of ASCII characters
- We denote this encoding with $\langle\rangle$ braces
  - The encoding must be unique for any object being encode
  - A string must be finite in length
- Examples:
  - The encoding of an integer could be itself as a string: $k = 376 \rightarrow \langle k \rangle = \texttt{376}$
    - We stress that this is the string with characters `3`, `7`, `6`
  - The encoding of a C++ program could be a string of its source code

Vocabulary Checkpoint

- In your own words, describe the relationship between
  - alphabet
  - input,
  - string, and
  - language

  of a decision problem

Empty Strings vs Empty Set

- Do not confuse between an empty string ε and an empty set $\emptyset$!
  - ε is a symbol used to represent a string of 0 length
  - $\emptyset$ is a set with no elements
- T/F: $\emptyset$ is a language
  - TRUE: Recall that a language is a *set*
- What about Σ^*, the set of all finite length strings?
  - TRUE

Decision Problems to Languages

Decision problem: "Does a given non-negative integer k have 3 as its last digit, when written in base 10?"

- A reasonable alphabet: $\Sigma = \{\texttt{0},\texttt{1},\dots,\texttt{9}\}$
- For example, if k is the integer forty-seven, then $\langle k \rangle = \texttt{47}$
  - We stress that this is a *string* of the characters $\texttt{4}$ and $\texttt{7}$, which represents the number forty-seven
- A corresponding language would be

$$L_{EndsWith3} = \{\langle k \rangle : k \text{ mod } 10 = 3\}$$

- It would also be acceptable to copy the phrasing of the decision problem, i.e.,

$$L_{EndsWith3}\{\langle k \rangle : k \text{ written in base 10 has 3 as the last digit}\}$$

Discuss: For the decision problem "Is a given base-10 integer x prime?" What is wrong with using $\Sigma = \mathbb{Z}$?

We need Σ to be finite

Decision Problems to Languages

Decision problem: "Is a given array of non-negative integers sorted?"

- A reasonable alphabet: $\Sigma = \{\texttt{0},\texttt{1},\dots,\texttt{9},\texttt{[},\texttt{,},\texttt{]}\}$
  - We need '`[`' and '`]`' to indicate the start and end of the array
  - We need '`,`' to separate the integers
- For example, if the array A has entries one, two, three, and four, then

$$\langle A \rangle = \texttt{[1,2,3,4]}$$

- A corresponding language is

$$L_{sorted} = \{\langle A \rangle : A \text{ is a sorted array of non-negative integers}\}$$

Deterministic Finite Automata

Notes

Deterministic Finite Automata (DFAs)

- A DFA (also called FSM) is a simple computational device, often drawn as a directed graph, that outputs "accept" or "reject" given an input

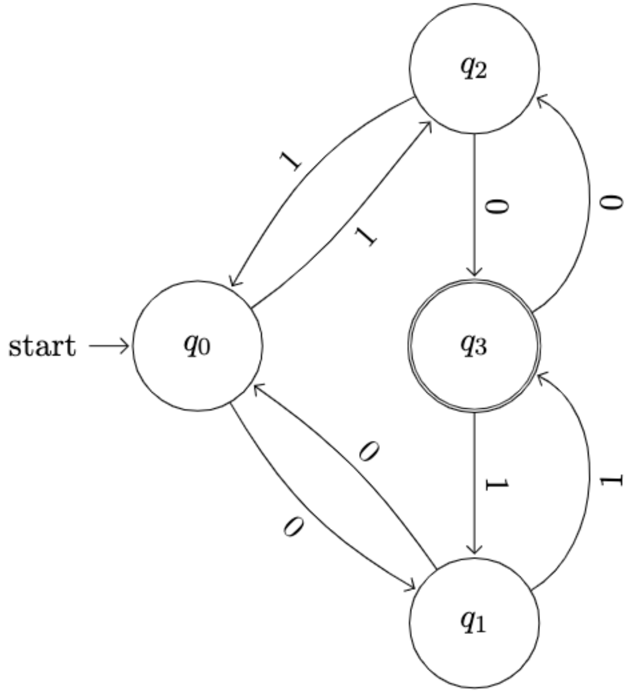
- It is defined by the five-tuple $M = (Q, \Sigma, \delta, q_0, F)$
  - Q is a finite set of states that make up its memory
  - DFAs takes an input string formed from the alphabet Σ, and view each character sequentially exactly once
  - δ is the transition function, if dictates which state the machine moves to next, given its current state and input characters
  - q_0 is the "start state"
  - F is the set of accept states

Five-tuple of DFA

- Full set of states $Q = \{q_0, q_1, q_2, q_3\}$
- Set of characters the DFA reads $\Sigma = \{0,1\}$
- Transition function for the DFA:

state	input	next state
q_0	0	q_1
q_0	1	q_2
q_1	0	q_0
q_1	1	q_3
q_2	0	q_3
q_2	1	q_0
q_3	0	q_2
q_3	1	q_1

- q_0: Start state. **Clearly indicate it with an arrow labelled start**
- Set of accept states, $F = \{q_3\}$



Deciding Languages with DFAs

- For language A, we say that DFA D decides A if and only if D:
  - Accepts for all $w \in A$
  - Rejects for all $w \notin A$

- A DFA takes input string $w = w_1 w_2 w_3 \dots w_n$ and runs as follows:

```
current_state = q_0                        ▷ We take q_0 to be the starting state by convention
i = 1                                      ▷ i is effectively a pointer to our position in w
while i ≤ n do
    current_state = δ(w_i, current_state)
    i = i + 1
                                           ▷ We have finished parsing the input
if current_state is an accept state then
    accept w
else
    reject w
```

  - Note: the DFA only accept/ reject after parsing the entire input!

String Formatting (Regular Expressions)

- We use regular expressions to represent the pattern of strings in a language
  - They're also widely used in computer programs to search for matching strings/ substrings
- We use the characters in the alphabet, and special characters $*, +, (\quad)$, and |
  - The parentheses () mean to treat whatever is inside as a single clause
  - The star * means to look for zero or more of the clause on the left
    - $(ab)^*$ matches $\varepsilon, ab, abababab,$ etc
  - The plus + means to look for one or more of the clauses on the left
    - j^+ matches $j, jjj, jjjjjjjjjjj,$ etc
  - The bar | means look for either the clause on the left or the clause on the right
    - $(CSE)|(ECE)$ matches 'ECE' or 'CSE'

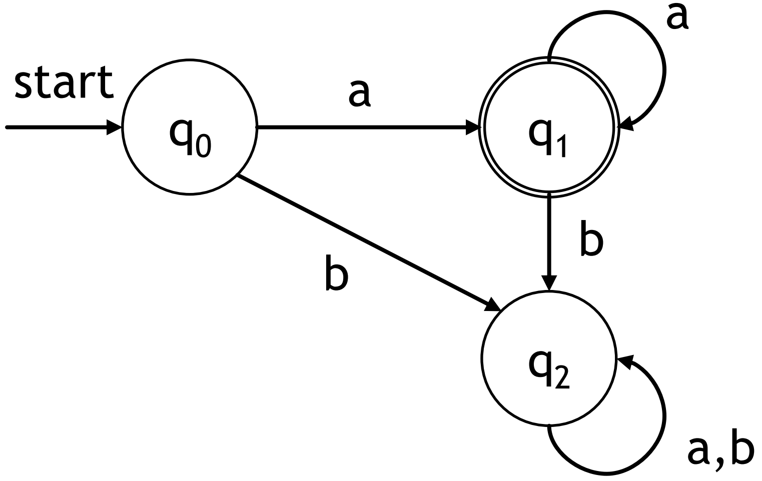
- $aabb(a|b)$ matches strings that start with two a's, followed by two b's, ending in either an a or a b. The only two strings that match this regular expression are $aabba$ and $aabbb$.

Infinitely many strings vs Infinite string

- Give some example of strings that matches $a^*(ab)^*$
  - $\varepsilon, a, ab, aaab, abab$
- How many strings are there that match this pattern?
  - Infinitely many
- Is it possible to have a string that matches $a^*(ab)^*$ that is infinite(-length)?
  - No! By definition, a string must be finite(-length)
  - Analogy: There are infinitely many natural numbers, and they grow arbitrarily large, but every individual natural number has some finite size

DFA Practice 1

- What language does this DFA decide over the alphabet {a,b}?

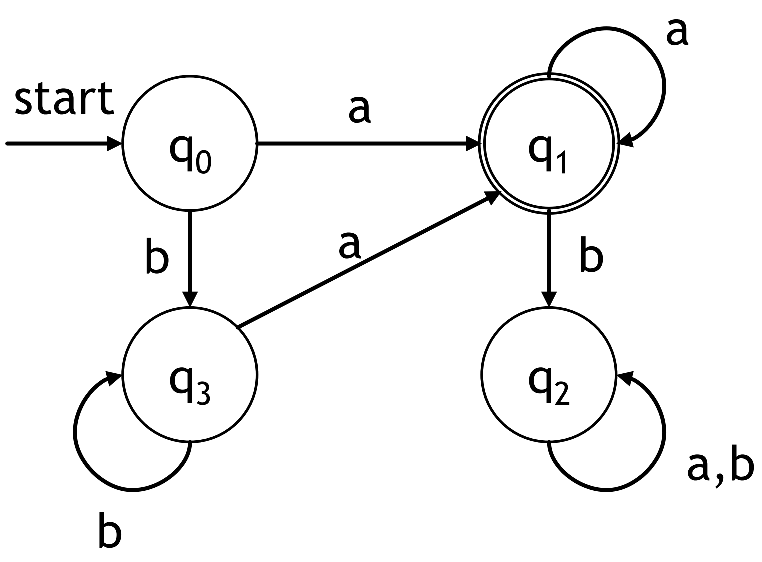


Observations:

- q_2 is a “sink” state
- q_1 is the accept step
  - Once we reach q_1, we enter q_2 as long as we get a ‘b’, but we can have any number of ‘a’ $\Rightarrow$ a* at the end
- We must start with an ‘a’, otherwise we enter q_2 immediately $\Rightarrow$ a at the start
- The regexp of the language is $aa^* = a^+$

DFA Practice 2

- What language does this DFA decide over the alphabet {a,b}?



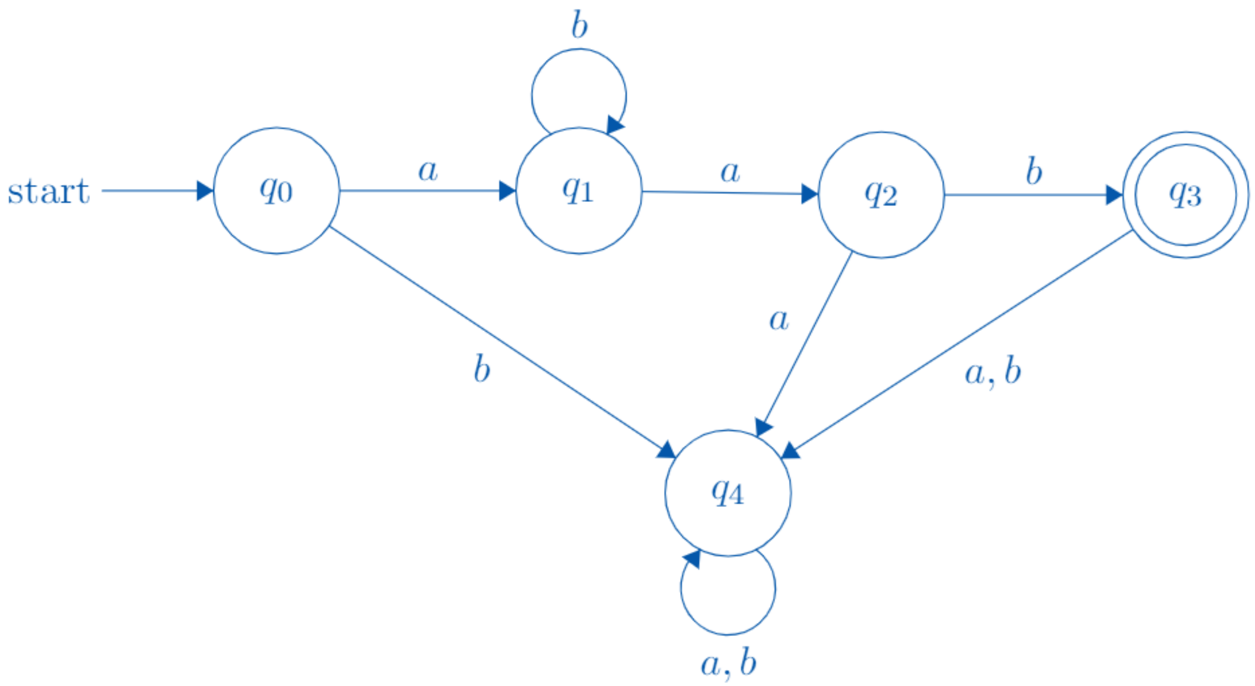
Observations:

- q_2 is a "sink" state
- a* at the end
- Path 1: $q_0 \rightarrow q_1$: a^+
- Path 2: $q_0 \rightarrow q_3 \rightarrow q_1$: ba^+
- Combining path 1 and 2: b is optional
  ⇒ b* at the start
- The regexp is b^*a^+

Language to DFA

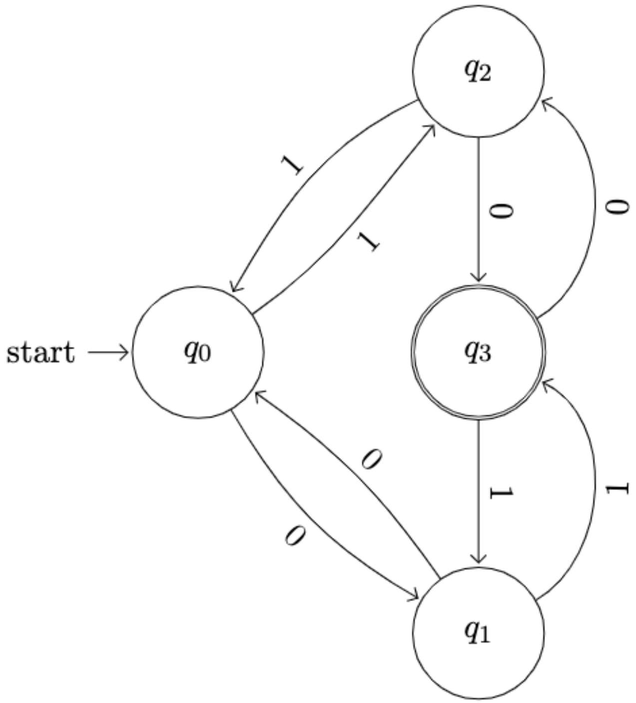
Draw a DFA that decides the language ab^*ab

Solution:



DFA to Language

- What language does this DFA decides?



$L = \{b \in \{0,1\}^*: b \text{ has odd number of } 0'\text{s and odd number of } 1'\text{s}\}$

Not all Problems are Decidable by DFAs!

- In lecture we showed the undecidability of $L_{01} = \{x \mid x \text{ matches } 0^n 1^n \text{ for } n \geq 0\}$

- Another language undecidable by DFAs is $L_{palindrome}$, the set of binary palindromes of arbitrary length
  - The idea is that for every new length of palindrome, the DFA would require an entirely new subset of states
  - If there are infinite lengths of palindromes, the DFA would need infinite states
  - This is not possible, as DFAs have finite memory
  - This language cannot be regular

- We think this is a pretty interesting result!
  (Don't worry about these types of proofs)