Regular Expressions -> Pattern matching

Substring search -> find a single string in text  
Pattern matching -> Find one of a **specified set** of strings in text.

Pattern matching applications:

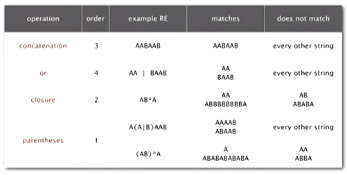
* Natural language processing
* Scan for virus signatures
* Filter text (spam, NetNanny, malware, etc.)
* Validate data-entry fields (dates, email, URL, credit card)

Parse text files

* Compile a Java program
* Crawl and index the Web
* Read in data stored in ad hoc input file format

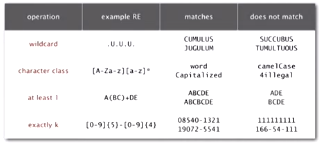
Regular expression: a notation to specify a set of strings.

Operations (below) are concatenation, or, closure, parenthesis



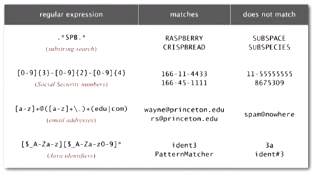
Additional operations are often added for convenience:

Operations (below) are wildcard, character class, at least 1, exactly k



RE notation is surprisingly expressive: you can use it for substring search

Applications of regex to substring search:



Regular expressions caveat:

Writing a RE is like writing a program:

* Need to understand programming model
* Can be easier to write than to read
* Can be difficult to debug

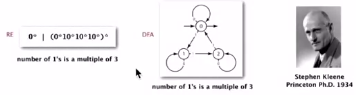
REs and DFAs

RE: concise way to describe a set of strings

DFA: Machine to recognize whether a given string is in a given set

Kleene’s theorem:

* For any DFA, there exists a RE that describes the same set of strings
* For any RE, there exists a DFA that recognizes the same set of strings



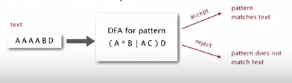
Overview is the same as for KMP

* No backup in text input stream
* Linear-time guarantee

Underlying abstraction: deterministic finite state automata (DFA)

Basic plan [apply Kleene’s theorem]

* Build DFA from RE
* Simulate DFA with text as input



INFEASIBLE! Basic plan is infeasible as DFA may have an exponential number of states.

How do we revise this? USE AN **NFA** instead of a DFA

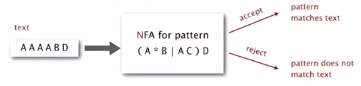
Overview *similar* to KMP:

* Still no backup in text input stream
* Now: **quadratic-time** guarantee

Underlying abstraction: **Non**deterministic finite state automata (NFA)

Basic plan [again with Kleene’s theorem]

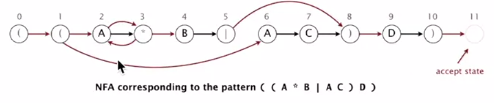
* Build NFA from RE
* Simulate NFA with text as input



Nondeterministic finite-state automata (NFA)

Regular-expression-matching NFA:

* RE enclosed in parenthesis
* One state per RE character (start = 0, accept = M)
* Red ε-transition (change state, but don’t scan text) [epsilon transition]
* Black match transition (change state and scan to next text char)
* Accept if any sequence of transitions (after scanning all text characters) ends in accept state

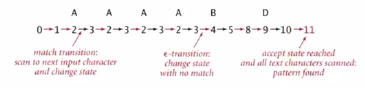


Nondeterminism:

* One view: machine can guess the proper sequence of state transitions
* Another view: sequence is a proof that the machine accepts the text

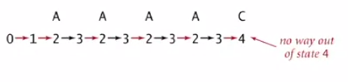
Examples

* The RE ‘A A A A B D’ is matched by the aforementioned NFA



Reason: there is some sequence of legal transitions that ends in state 11  
 [even though some sequences end in wrong state or stall]

* The RE ‘A A A C’ is not matched by the aforementioned NFA



Reason: there is **no** sequence of legal transitions that ends in state 11?  
 [but you must argue about all possible sequences]

--

Determining whether a string is matched by an automaton:

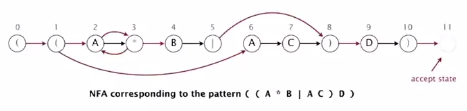
* DFA: since it is deterministic, this is easy. There is only one applicable transition
* NFA: since it is nondeterministic and there can be several applicable transitions, you must select the right one
  + **Must simulate NFA** by systematically considering all possible transition sequences

NFA simulation

State names: integers from 0 to *M.* (M is number of symbols in RE)

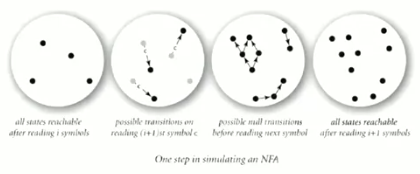
Match transitions: keep regular expression in array re[]

ε-transitions: store in a digraph G  
0 -> 1, 1 ->2, 1 -> 6, 2 -> 3, 3 -> 2, 3 -> 4, 5 -> 8, 8 ->9 , 10 -> 11



How to efficiently simulate an NFA:

Maintain a set of all possible states that NFA could be in after reading the first *i* text characters



Q: How to perform reachability? (how do we find all possible states after epsilon transitions)

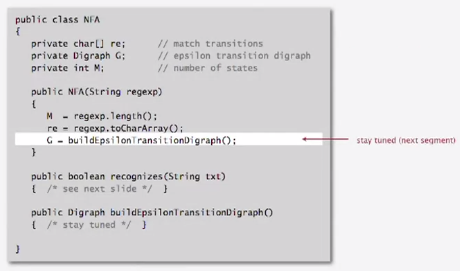
Digraph reachability

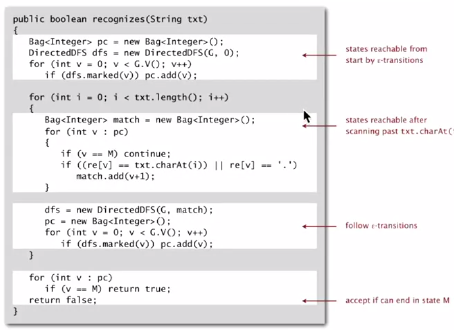
**Digraph reachability:** find all vertices reachable from a given source or set of vertices

Public class DirectedDFS  
DirectedDFS(Digraph G, int s) : find vertices reachable from s  
DirectedDFS(Digraph G, Iterable<Integer> s) : find vertices reachable from sources  
boolean marked(int v) : is v reachable from source(s)?

Reachability solution: run DFS from each source, without unmarking vertices

Performance: runs in time proportional to E + V (linear)

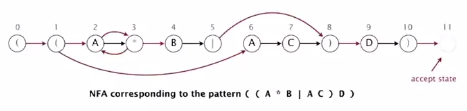




NFA simulation analysis

Proposition: determining whether an N-character text is recognized by the NFA corresponding to an M-character pattern takes time proportional to M N in the worst case.

Proof: for each set of the N text characters, we iterate through a set of states of size no more than M and run DFS on the graph of ε-transitions  
 [The NFA construction we will consider ensures that the number of edges <= 3M, as no node has more than 3 edges leaving it- see below]



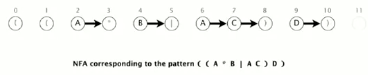
NFA construction - Building an NFA corresponding to an RE

**States:** Include a state for each symbol in the RE, plus an accept state

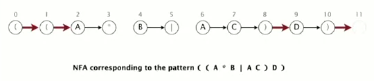
Concatenation: Add match-transition edge from state corresponding to characters in the alphabet to next state.

Character types are below:

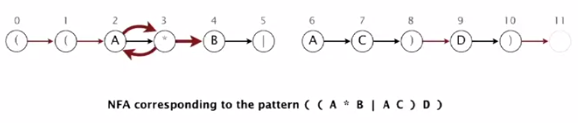
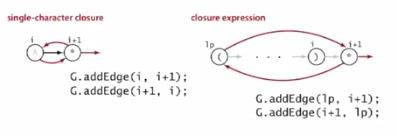
* Alphabet: A B C D
* Metacharacters: ( ) , \* |



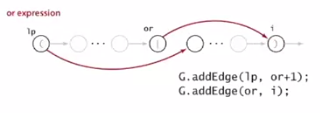
Parenthesis: add ε-transition edge from parenthesis to next state

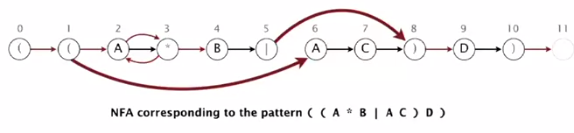


Closure: add three add ε-transition edges for each \* operator as below:



Or: Add two ε-transition edges for each | operator as below:





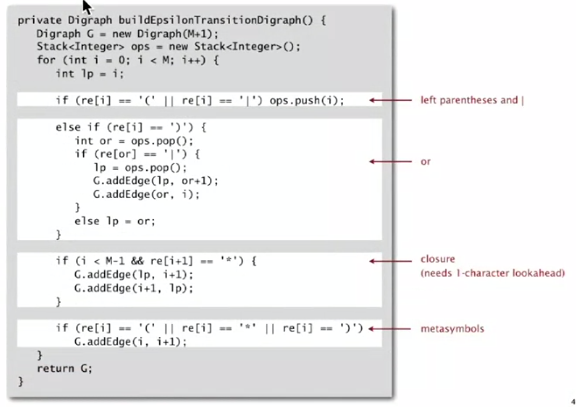
Goal: write a program to build the ε-transition digraph

Challenges: remember left parentheses to implement closure and or; remember | to implement or.

Solution: Maintain a stack

* ( symbol: push ( onto stack
* | symbol: push | onto stack
* ) symbol: pop corresponding ( and possibly intervening |;  
  add ε-transition edges for closure/or

NFA construction Java implementation

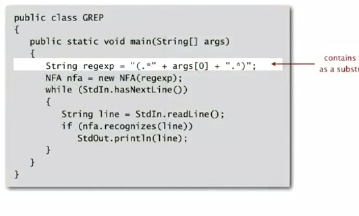


Proposition: building NFA corresponding to an M-character RE takes time and space proportional to M

Proof: For each of the M characters in the RE, we add at most three ε-transitions and execute at most two stack transitions

RE Applications

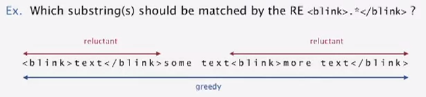
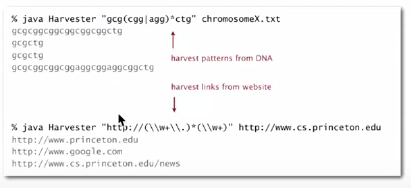
* Grep: Take a RE as a command-line argument and print the lines from standard input having some substring that is matched by the REB



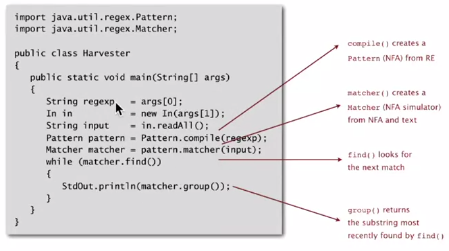
Bottom line: worst-case for the grep (proportional to M N) is the same for brute-force substring search

----> Grep application example: Crossword puzzles

Industrial-strength grep implementation (the one taught here is not industrial strength) would include:

* Add wildcard
* Add multiway or
* Handle metacharacters
* Support character classes
* Add capturing capabilities (give me the substring that matches)
* Extend the closure operator
* Error checking and recovery
* Greedy vs reluctant matching
* Harvesting information: Print all substrings of input that match a RE

Java’s java.util.regexp.Pattern and java.util.regexp.Matcher are two classes that can be used for harvest:



WARNING re: algorithmic complexity attacks

Typical implementations (e.g. Unix grep, Java, Perl) do not guarantee performance! 

Example: SpamAssassin regular expression

* Takes exponential time on pathological email address
* One can use such addresses to DOS a mail server



Not-so-regular expressions:

Back-references

* \1 notation matches subexpression that was matched earlier
* Supported by typical RE implementations



Some non-regular languages:

* Strings of the form *w w* for some string *w:* beriberi
* Unary strings with a composite number of 1s: 111111
* Bitstrings with an equal number of 0s and 1s: 01110100
* Watson-Crick complemented palindromes: atttcggaaat

If non-regular, Kleene’s theorem doesn’t hold (and no NFA corresponds)

Therefore: Pattern matching with back-references is INTRACTABLE.

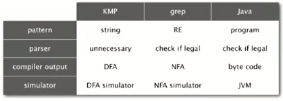
Context

Abstract machines, languages and nondeterminism:

* Basis of the theory of computation
* Intensively studies since the 1930s
* Basis of programming languages

Compiler: a program that translates a program to machine code

* KMP: string -> DFA
* Grep: RE -> NFA
* Javac: Java language -> Java byte code



---

Summary of pattern-matching algorithms:

Programmer:

* Implement substring search via DFA simulation
* Implement RE pattern matching via NFA simulation

Theoretician:

* RE is a compact description of a set of strings
* NFA is an abstract machine equivalent in power to RE
* DFAs and REs have limitations

Student: practical application of core computer science principles…

Example of essential paradigm in computer science:

* Build intermediate abstractions
* Pick the right ones
* Solve important practical problems