COMP4019 - Solutions 4 – Red-Black Trees; Dynamic Programming

Xavier Carpent & Ian Knight

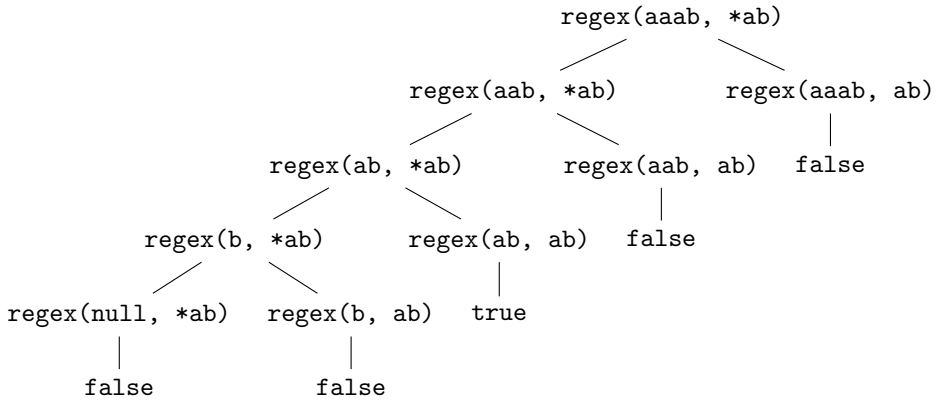
October 28, 2021

Dynamic Programming - Regex Matching

The idea of the algorithm `regex(s, p)` is going to be as follows. The strings `s` and `p` are divided into head (first character) and tail (rest of the string): `s` = `hs|ts` and `p` = `hp|tp`. At each step, we compare the two heads:

- if `hs` = `hp`, we return `regex(ts, tp)`
- if `hs` ≠ `hp`, we return `false`
- if `hp` = `?`, we return `regex(ts, tp)`
- if `hp` = `*`, we branch and return `true` iff either returns `true`:
  - `regex(ts, p)` (matches at least 1 character)
  - `regex(s, tp)` (matches 0 character)

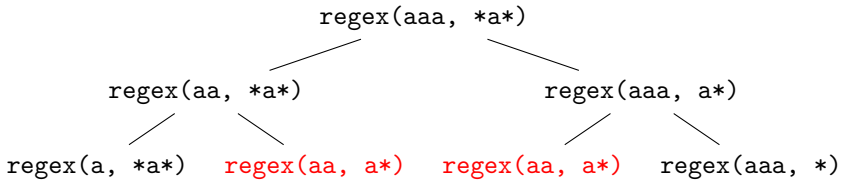
If `ts` is `null`, and `tp` is either `null`, or `*`, we return `true`.
Here is an example, which returns `true`:



The central problem in this simplified regex matching is the `*` character, which makes the number of cases explose.

1. • Optimal Substructure: we divide the full problem into sub-problems; the answer to the full problem is a composition of the answers of the sub-problems;

- Overlapping Subproblems: many subproblems are in common; consider the following example: `s` = `aaa` and `p` = `*a*`:



  The central sub-trees contain the same sub-problems.

2. Simple implementation in Python:

```python
def regex(s, p):
  if not s:
    if not p:
      return True
    elif p[0] == '*':
      return regex(s, p[1:])
    else:
      return False
  if not p:
    return False

  if p[0] == '?':
    return regex(s[1:], p[1:])
  elif p[0] == '*':
    return regex(s[1:], p) or regex(s, p[1:])
  elif p[0] == s[0]:
    return regex(s[1:], p[1:])
  else:
    return False

print(regex('aaab', '*ab'))
print(regex('aaab', '*abc'))
print(regex('aaa', '*a*'))
print(regex('hello', '?e*o'))
```

  Dynamic programming approach (identical, but with memoization). In practice, you would use a `@lru_cache` decorator or equivalent for better readability, but this is not a programming class...

```python
regex_dp_dict = {}

def regex_dp(s, p):
  if (s, p) in regex_dp_dict:
    return regex_dp_dict[(s, p)]

  if not s:
    if not p:
      ans = True
      regex_dp_dict[(s, p)] = ans
      return ans
    elif p[0] == '*':
      ans = regex_dp(s, p[1:])
      regex_dp_dict[(s, p)] = ans
      return ans
    else:
      ans = False
      regex_dp_dict[(s, p)] = ans
      return ans
```

```
20
21  if not p:
22    ans = False
23    regex_dp_dict[(s, p)] = ans
24    return ans
25
26  if p[0] == '?':
27    ans = regex_dp(s[1:], p[1:])
28    regex_dp_dict[(s, p)] = ans
29    return ans
30  elif p[0] == '*':
31    ans = regex_dp(s[1:], p) or regex_dp(s, p[1:])
32    regex_dp_dict[(s, p)] = ans
33    return ans
34  elif p[0] == s[0]:
35    ans = regex_dp(s[1:], p[1:])
36    regex_dp_dict[(s, p)] = ans
37    return ans
38  else:
39    ans = False
40    regex_dp_dict[(s, p)] = ans
41    return ans
```

Complexity: proportional to the number of new "nodes" in the call tree. Since each iteration decreases the size of `s` OR `p` by one, the number of such nodes is bounded by $|s||p|$, the complexity is thus $O(|s||p|)$. The memory is the same (each "length" of `s` and `p` is saved).