5. Regular expression matching

Given a string S and a regular expression R, write a function to check if the S matches the regular expression R. S only contains letters and numbers.

A regular expression consists of

1. Letters A-Z

2. Numbers 0-9

3. ‘*’ - Matches 0 or more characters.

4. ‘.’ - Matches one character.

For example:

S=“ABBBAC”

R = “.*A*”

Return true

S=“GREATS”

R = “G*T*E”

Return false

1. State

Parameters

i - last index of substring in S,

j - last index of substring in R,

Cost function

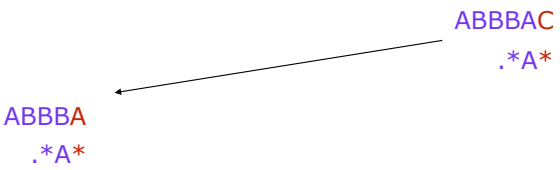
matches(i,j,S,R) - return true , if substring of S ending at i, matches substring of regular expression R ending at j.

Regular expression matching

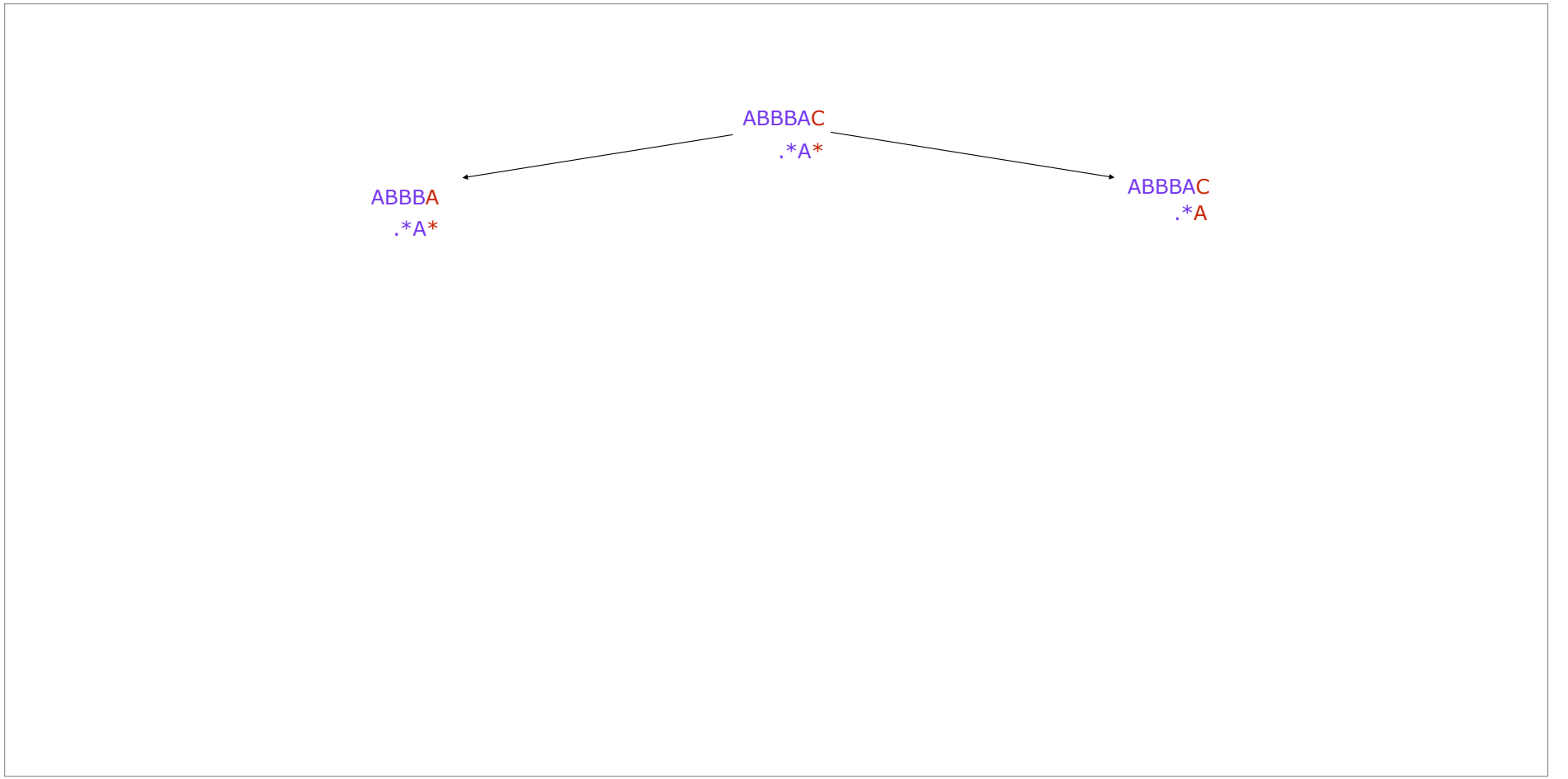
ABBBAC

.*A*

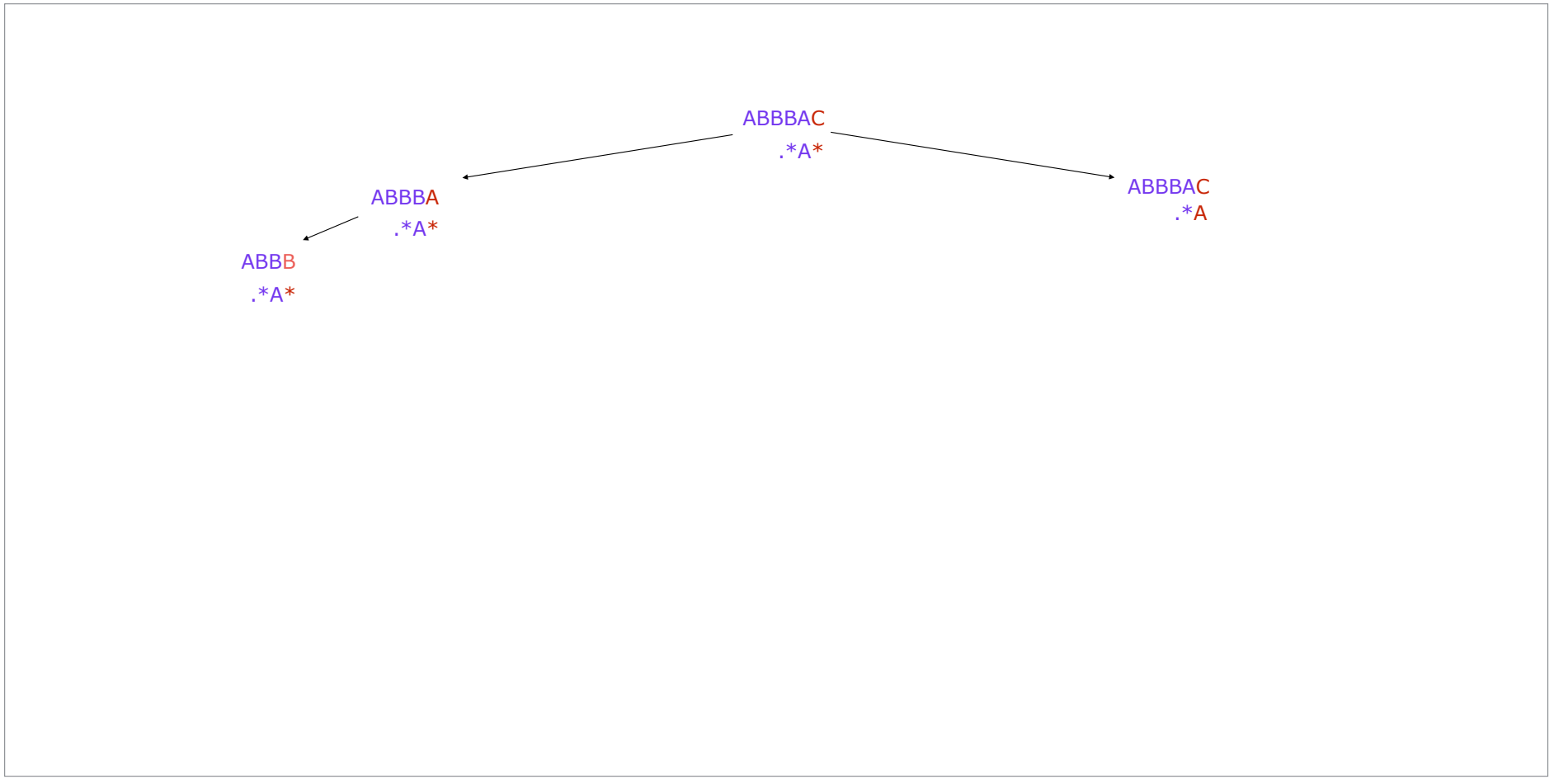
Regular expression matching



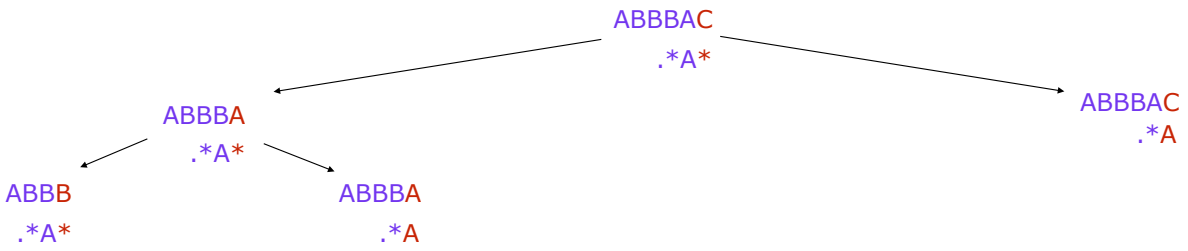
Regular expression matching



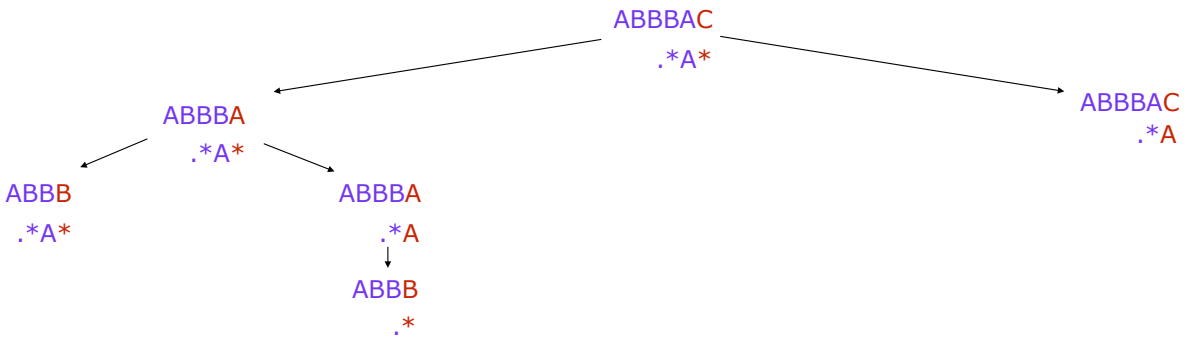
Regular expression matching



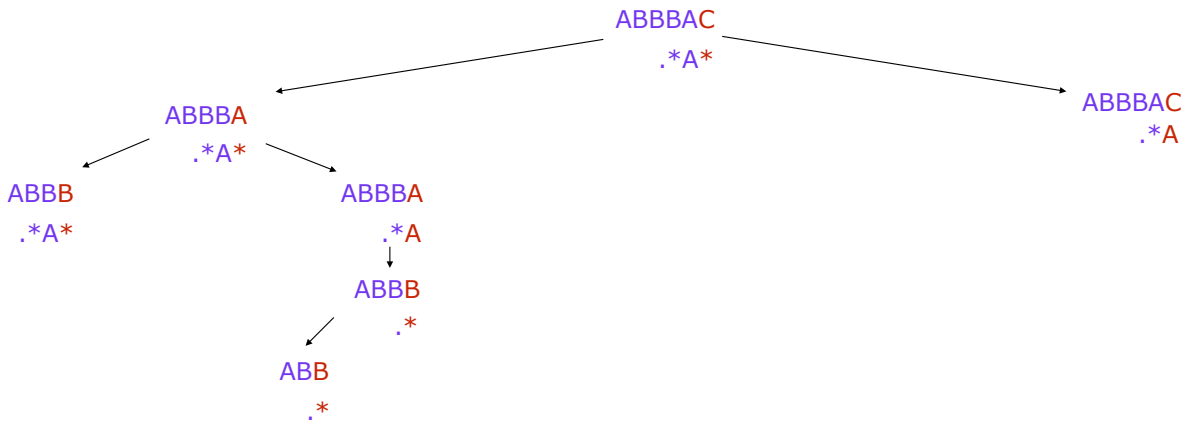
Regular expression matching



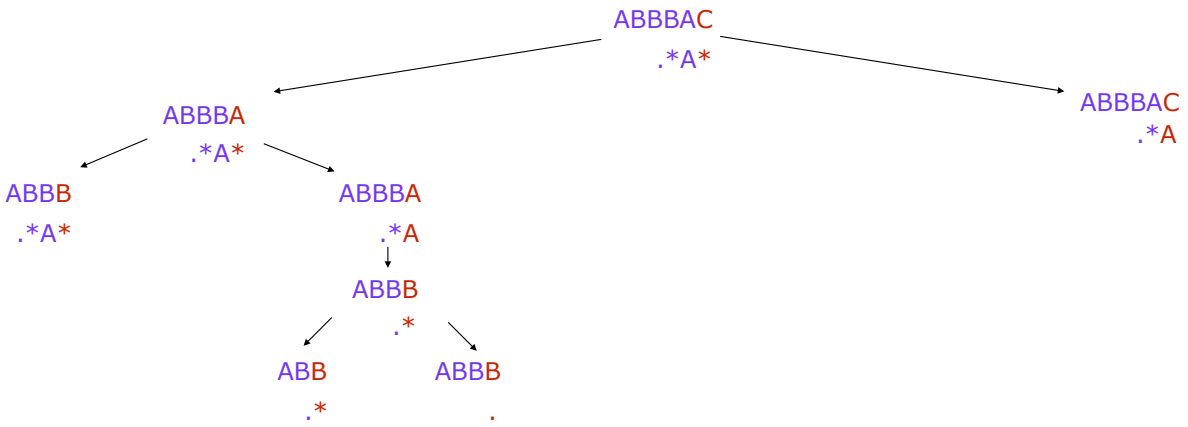
Regular expression matching



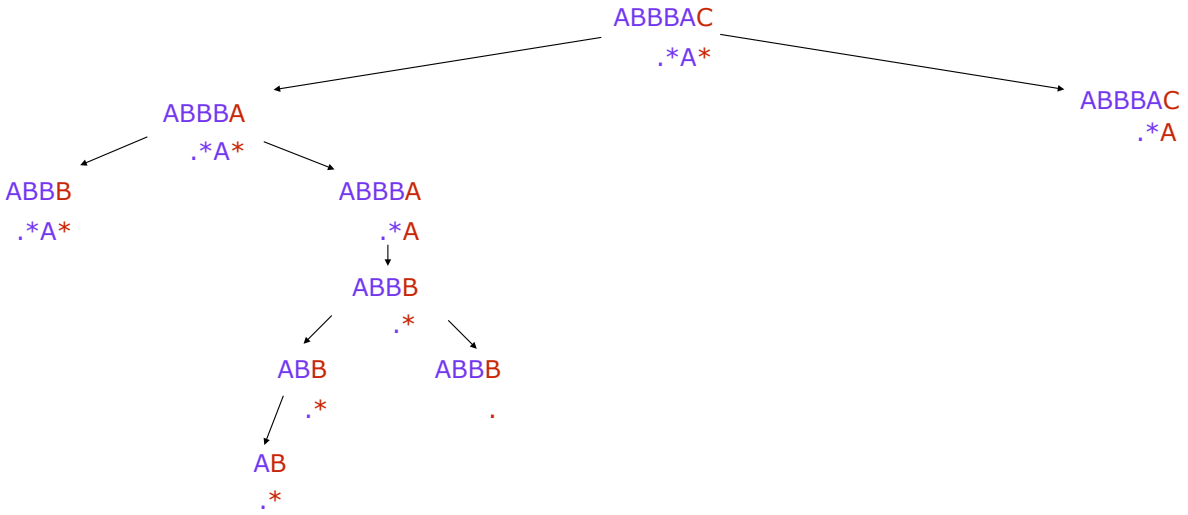
Regular expression matching



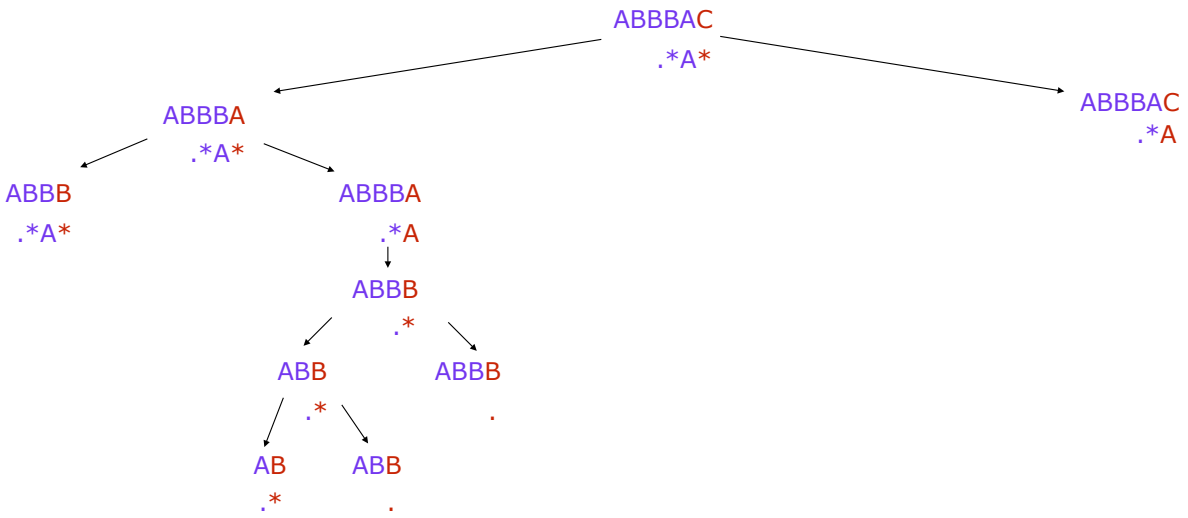
Regular expression matching



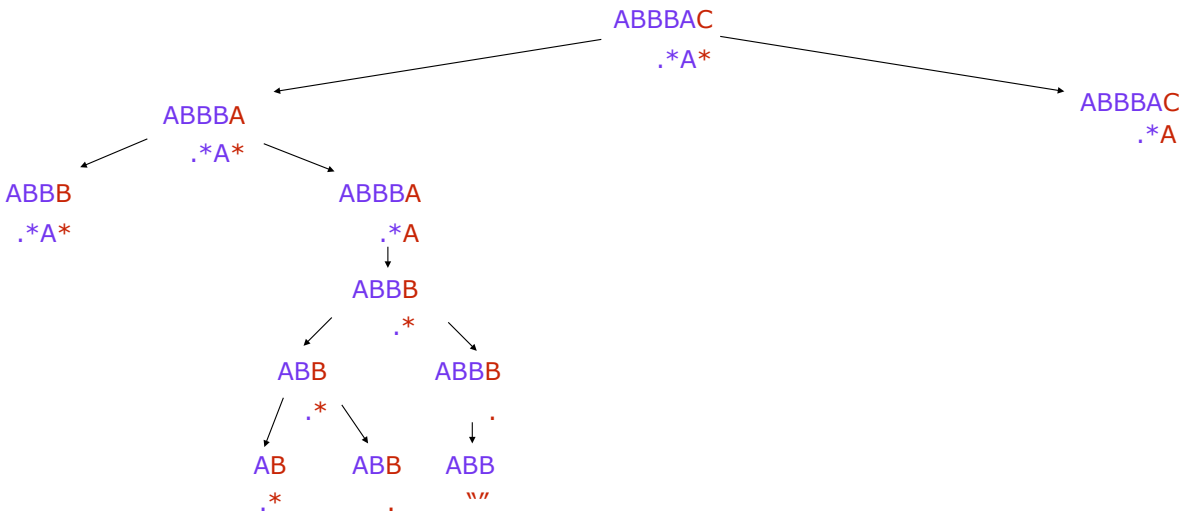
Regular expression matching



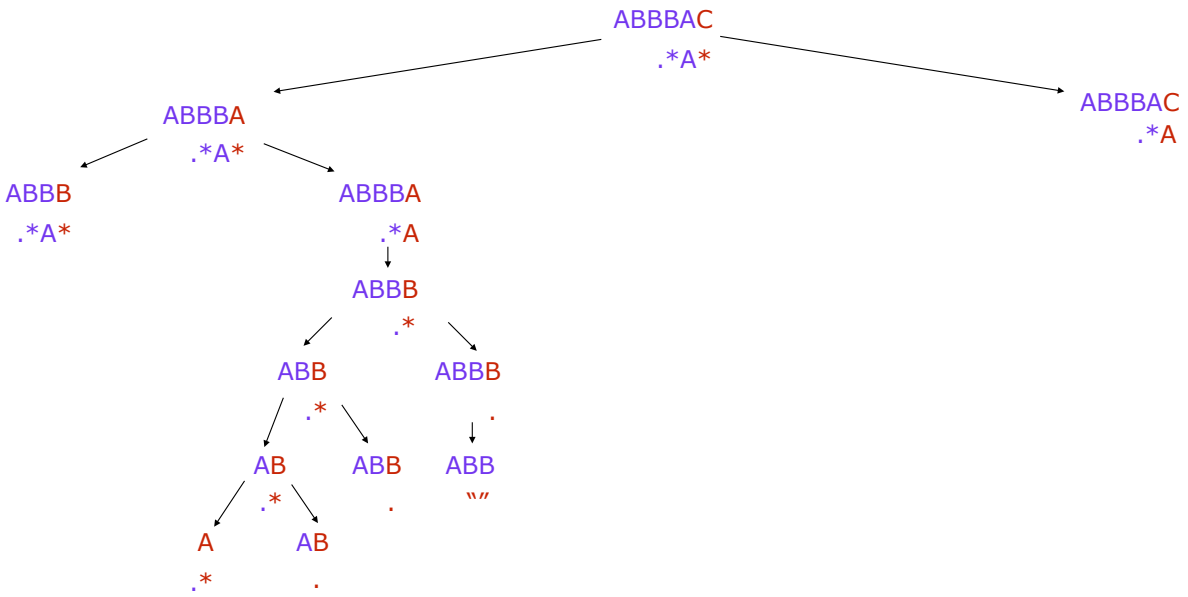
Regular expression matching



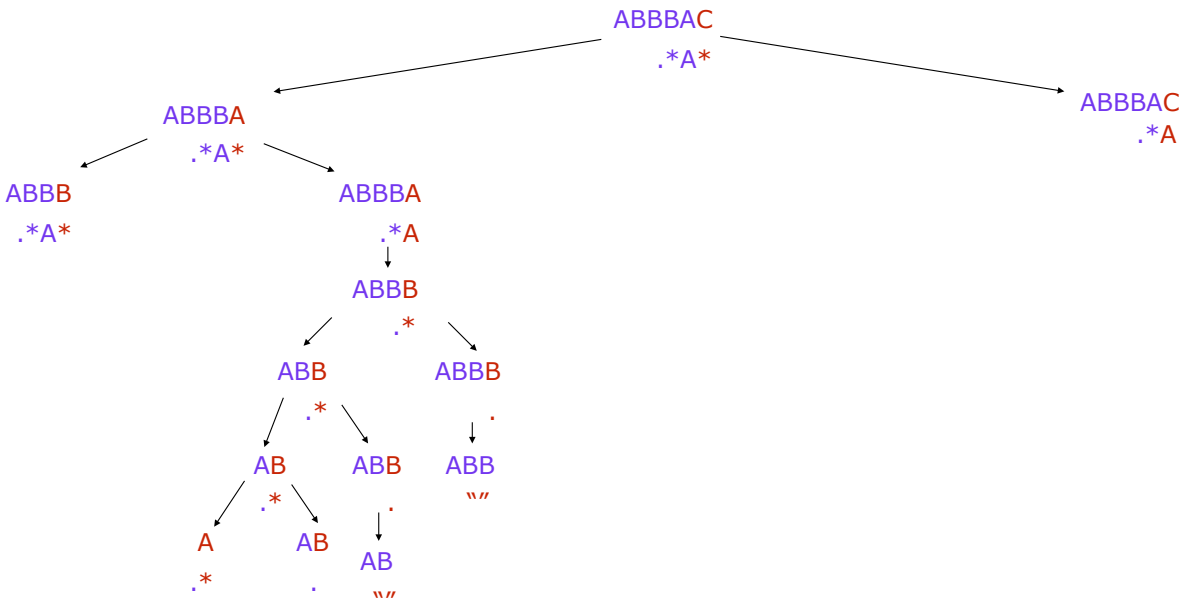
Regular expression matching



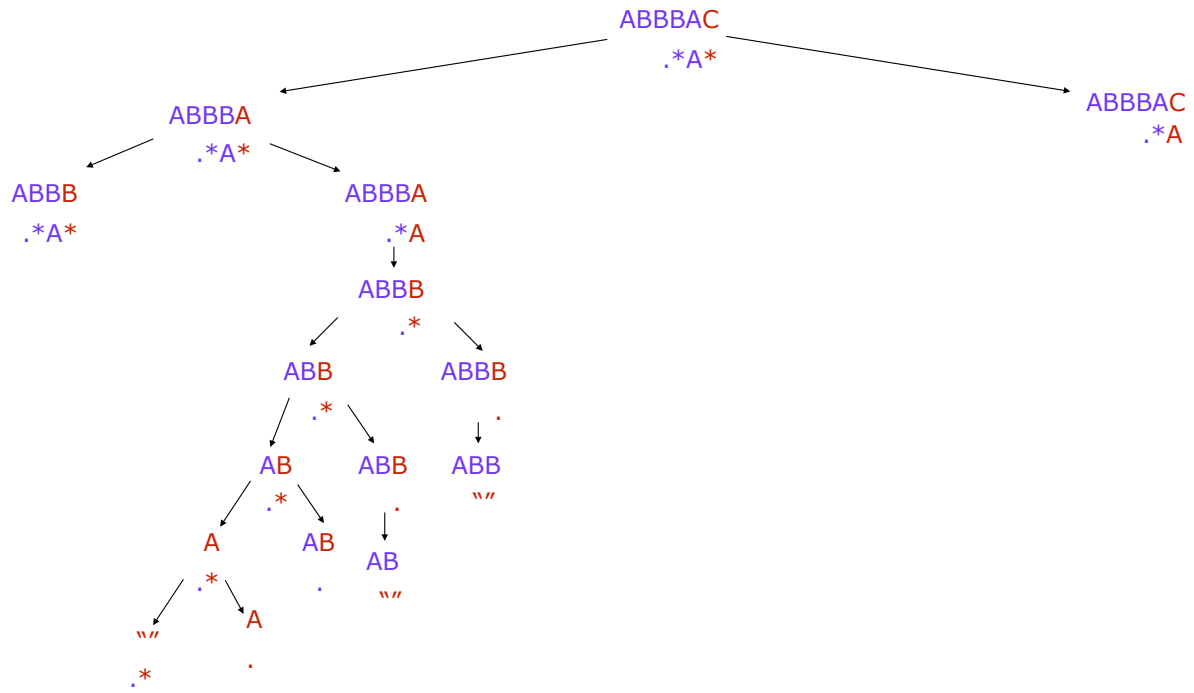
Regular expression matching



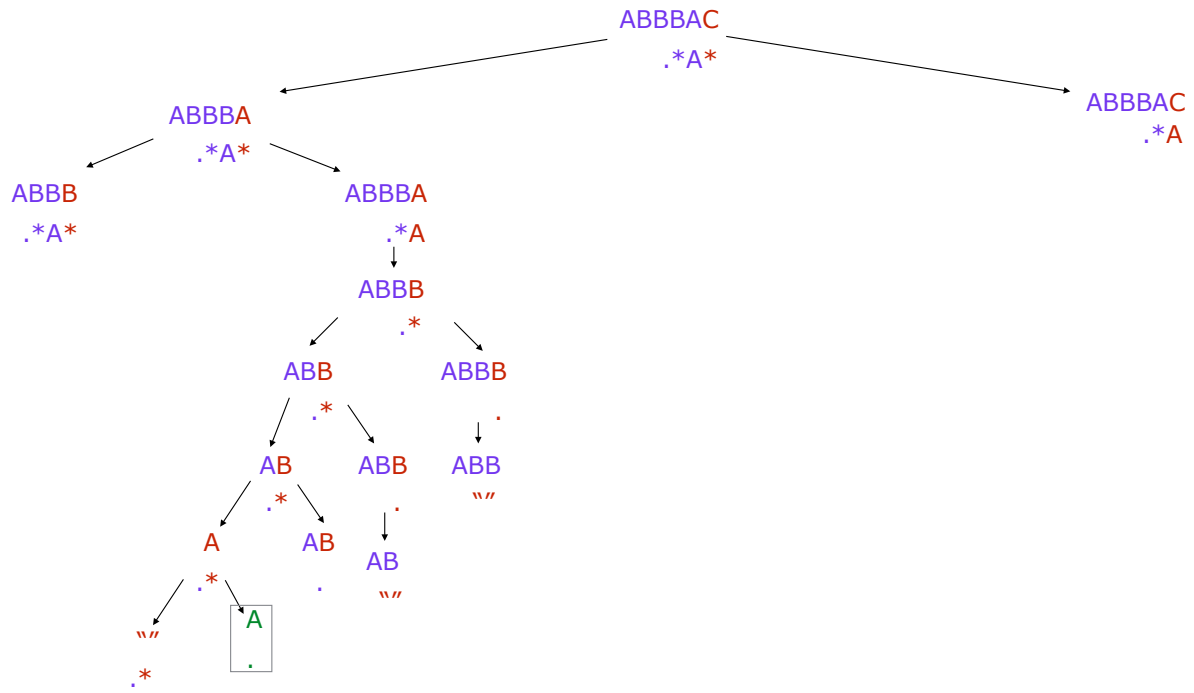
Regular expression matching



Regular expression matching



Regular expression matching



2. Transitions

Base case

i=-1 and j=-1, return true, all the characters matched and both S and R are empty.

i>=0 and j=-1, return false, some characters in S are unmatched

i=-1 and j>=0 , return false, some characters in R are unmatched

R[0:j] = '*', return true, * matches everything in S

Case 1:

R[j] = S[i], then

matches(i,j) = matches(i-1,j-1)

Case 2:

R[j] = '.'

matches(i,j) = matches(i-1,j-1)

Case 3:

R[j] = '*'

matches(i,j) = matches(i-1,j) or matches(i,j-1)

Recurrence relation

matches(i,j,S,R) = True , if i=-1, j=-1

matche(i,j,S,R) = False , if i=-1 or j=-1

matches(i,j,S,R) = matches(i-1,j-1), if S[i] = R[j]

matches(i,j,S,R) = matches(i-1,j-1), if R[j] = .

matches(i,j,S,R) = matches(i-1,j) or matches(i,j-1), if R[j] = *

3. Recursive solution

Java

```java
public static boolean matches(int i, int j, String S, String R) {
    if ((i == -1 && j == -1) || R.substring(0, j + 1).equals("*"))
{
        return true;
    } else if (i == -1 || j == -1) {
        return false;
    }
    if (S.charAt(i) == R.charAt(j)) {
        return matches(i - 1, j - 1, S, R);
    } else if (R.charAt(j) == '.') {
        return matches(i - 1, j - 1, S, R);
    } else if (R.charAt(j) == '*') {
        return matches(i - 1, j, S, R) || matches(i, j - 1, S, R);
    }
    return false;
}
```

Java

```java
public static boolean matches(String S,String R){
    int M = S.length();
    int N = R.length();
    return matches(M-1,N-1,S,R);
}
```

Python

```
def matches(i, j, S, R):
    if (i == -1 and j == -1) or R[0:j + 1] == "*":
        return True
    if i == -1 or j == -1:
        return False
    if S[i] == R[j]:
        return matches(i - 1, j - 1, S, R)
    elif R[j] == '.':
        return matches(i - 1, j - 1, S, R)
    elif R[j] == '*':
        return matches(i - 1, j, S, R) or matches(i, j - 1, S,
R)
    return False
```

4. Memoize

4. Memoization

We can cache the results in a 2D array

Key -> i,j

Value -> True if the string S[0:i] matches the regular expression R[0:j]

Java

```java
public static boolean matchesMemo(int i, int j, String S, String R, Boolean[][] cache) {
    if ((i == -1 && j == -1) || R.substring(0, j + 1).equals("*")) {
        return true;
    } else if (i == -1 || j == -1) {
        return false;
    }
    if (cache[i][j] != null) {
        return cache[i][j];
    }
    if (S.charAt(i) == R.charAt(j)) {
        cache[i][j] = matchesMemo(i - 1, j - 1, S, R, cache);
        return cache[i][j];
    } else if (R.charAt(j) == '.') {
        cache[i][j] = matchesMemo(i - 1, j - 1, S, R, cache);
        return cache[i][j];
    } else if (R.charAt(j) == '*') {
        cache[i][j] = matchesMemo(i - 1, j, S, R, cache) || matchesMemo(i, j - 1, S, R,
cache);
        return cache[i][j];
    }
    return false;
}
```

Java

```
public static boolean matchesMemo(String S,String R){
    int M = S.length();
    int N = R.length();
    Boolean[][] cache = new Boolean[M+1][N+1];
    return matchesMemo(M-1,N-1,S,R,cache);
}
```

Python

```python
def matches_memo(i, j, S, R, cache):
    if (i == -1 and j == -1) or R[0:j + 1] == "*":
        return True
    if i == -1 or j == -1:
        return False
    if cache[i][j] is not None:
        return cache[i][j]
    if S[i] == R[j]:
        cache[i][j] = matches_memo(i - 1, j - 1, S, R, cache)
        return cache[i][j]
    elif R[j] == '.':
        cache[i][j] = matches_memo(i - 1, j - 1, S, R, cache)
        return cache[i][j]
    elif R[j] == '*':
        cache[i][j] = matches_memo(i - 1, j, S, R, cache) or \
                      matches_memo(i, j - 1, S, R, cache)
        return cache[i][j]
    return False
```

5. Bottom up approach

5. Bottom up approach

Recurrence relation

matches(i,j,S,R) = True , if i=-1, j=-1

matche(i,j,S,R) = False , if i=-1 or j=-1

matches(i,j,S,R) = matches(i-1,j-1), if S[i] = R[j]

matches(i,j,S,R) = matches(i-1,j-1), if R[j] = .

matches(i,j,S,R) = matches(i-1,j) or matches(i,j-1), if R[j] = *

Bottom up equation

dp[i][j] = True , if i=0, j=0

dp[i][j] = False , if i=0 or j=0

dp[i][j] = dp[i-1][j-1], if S[i-1] = R[j-1]

dp[i][j] = dp[i-1][j-1] , if R[j-1] = .

dp[i][j] = dp[i-1][j] or dp[i][j-1], if R[j-1] = *

Java

```java
public static boolean matchesDP(String S,String R){
    int M = S.length();
    int N = R.length();
    boolean[][] dp = new boolean[M+1][N+1];
    dp[0][0] = true;
    for(int i=1;i<=M;i++){
        for(int j=1;j<=N;j++){
            if(S.charAt(i-1) == R.charAt(j-1)){
                dp[i][j] = dp[i-1][j-1];
            }else if(R.charAt(j-1) == '.'){
                dp[i][j] = dp[i-1][j-1];
            }else if(R.charAt(j-1) == '*'){
                dp[i][j] = dp[i-1][j] || dp[i][j-1];
            }
        }
    }
    return dp[M][N];
}
```

Python

```python
def matches_dp(S, R):
    M = len(S)
    N = len(R)
    dp = [[False for _ in range(N + 1)] for _ in range(0, M +
1)]
    dp[0][0] = True
    for i in range(1, M + 1):
        for j in range(1, N + 1):
            if S[i - 1] == R[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            elif R[j - 1] == '.':
                dp[i][j] = dp[i - 1][j - 1]
            elif R[j - 1] == '*':
                dp[i][j] = dp[i - 1][j] or dp[i][j - 1]
    return dp[M][N]
```

Memoization performs much better than bottom up approach.

Reason ?

It solves the subproblems on demand.