# 15-210 Assignment DPlab

Roy Sung
roysung@andrew.cmu.edu
Section E
4/25/2014

---

**5: Dynamic Programming**

---

**Task** 4.1

**Recursive Solution**

```
fun HTS(S) =
    let
    fun HT'(j) = if(j >= length(S)
                   then false
                   else  if(isWord(substring(0,j))
                         then HTS(substring(j,lengthS)
                         else HT'(j+1)
    in
    HT'(1)
    end
```

**Sharing**
This is the same as the sample solution given. Since j can range from 1 to 0, and j is the place hold to signal the next substring to take it represents both the end and the begining of the substring. Thus the number of calls is at most $n^2$, representing all the contingous substrings of $S$ that are words. If the substrings are words then they can be combined to together to make sentence thus the smaller substrings can be shared.

**DAG Costs**
Since we call HT at most $n^2$ distinct calls, and each HT does constant work in each level of recursion, the work of HT should have $n^2$ nodes. However, HTS basically splits the substring down into a smaller chunk of the original string, thus, the smaller substrings are just portions of the larger string. This means that at some point we have to reach the end of the string, or the empty string to be exact. This shows that the end depth of each recursive call will be at most $logn$. Since the substring we pass off is the just the backend of the original string, there are at most $n$ recursive calls can be made on the original string. Thus the span is $nlogn$. Thus the work of HT' is $O(n^2)$ and span of $O(nlogn)$.

**Task 4.2**
**Recursive Solution**

```
fun MkBridges (S1,S2) =
    let
        fun Brid = ((S1,S2),i) =
```

```
        let
          val first = nth S1 i
          val findFirstS2 = find(first,S2) (*returns the first index at
                                        which the element first is found in S2 *)
        in
          1 + findMax(S1<i,length(S1)>,S2<findFirstS2,length(S2)>)
        end
        fun findMax (S1,S2) =
            if(S1 is empty or S2 is empty)
            then 0
            else max(Brid(S1,S2,i) where i ranges from one to length(S1)))
    in
    findMax(S1,S2)
    end
```

**Sharing**

findMax has at $n$ calls since we just feed it contingous subsequences of the original two sequences. Within each we call at most $n$ times to Brid. Thus there will be at most $n^2$ distinct calls made. These are distinct calls because essentially we are just taking subseq of the original sequence. Thus the subsequences can be shared across the original subsequence in determining the maximum number of bridges that can be made.

**DAG and Costs**

The DAG for Brid has $n$ nodes of the sequence that are passed off to it. Since we passing off a portion of the second sequence, the DAG of Brid will have a depth of $logn$. Thus the work of Brid is $O(n)$ and the span is $O(logn)$. The DAG of find max has $n$ nodes. There and as said before it each node calls Brid $n$ times. Thus since the work of Brid is $O(n)$ and $O(logn)$, we thus work of findMax is $O(n^2)$ and span of $O(nlogn)$.

**Task 4.3**

**Recursive Solution**

```
fun findMin (A,S,R) =
    let
    fun stepThrough (a,(r1,r2,r3),i) =
        if( nth a i = r1)
        then apply((r1,r2,r2),i,a) returns a string that applied the rule
                                        (r1,r2,r3) to a at position i
        else a
    % s is some sequence, r is a rule and numSteps is the number of steps so far
    fun eachOne (s,subS,r) =
        if(s = S)
        then 0
        else if(length(s) >= length(subS))
             then Int.maxInt
             else
         %    all the stepThrough that return the same sequence a
                      that is inputed, will be disregarded
             1 + min(findMin'(stepThrough(s,r,i) such that each i is a index of s)
                                                    (subS)(r))
```

```
fun findMin' (a)(s)(r) =
    if(r = length(R))
    then Int.maxInt
    else min(eachOne(a,S,nth R r),findMin'(a,S,r+1))
in
findMin'  (1,S,0)
end
```

**Sharing**

The subproblems to this problem will be the the sequences larger than one and finding the minimum number of steps to reach the original goal sequence. findMin' will be called $mnk$. This is because in our code we apply a single rule for at most all the letters in the alphabet. We keep applying the rule until the rule can't be applied any more or there we have reached a dead end. These are distinct calls as when we apply a rule to a sequence we apply the rule to single character in that sequence. Thus if we get the same subsequence at another point in our algorithm we can check to see if there was already a min number of steps to get to the goal string **DAG and Costs**

The DAG of findMin' will have $n^2$ nodes. This is because findMin essentially gets passed all the sequences of size $n$. There are $n^2$ sequences of those and at each node findMin will do $O(mnk)$ work as explained above to get to the goal string. Thus the cost will be $O(mn^3k)$.