# CS325 Winter 2013: HW 5

Daniel Reichert
Trevor Bramwell

March 1, 2013

## 1

**Problem:** We will review proof by induction in this problem. Specifically, consider the stoogesort algorithm for sorting and use proof by induction to show that this algorithm correctly sorts an given array into increasing order.

```
STOOGESORT(A[0...n-1])
    if n = 2 and A[0] > A[1]
        swap A[0] and A[1]
    else if n > 2
        k = 2n/3
        STOOGESORT(A[0...k-1])
        STOOGESORT(A[n-k...n-1])
        STOOGESORT(A[0...k-1])
```

### Proof By induction:

*Proof.* Base case: $n = 2$. If $n = 2$ and $A[0] = 2$ and $A[1] = 1$, then the algorithm if statement will evaluate to true. This will cause $A[0]$ and $A[1]$ to be swapped, and the array is now sorted in increasing order proving our base case.
Inductive Step: Assume Stoogesort is true for $2...K-1$. Prove Stoogesort is true for $K$.
When stoogesort has a input greater than 2, it calls stoogesort($2K/3$). Since $2K/3$ is within our Inductive Assumption, we know that stoogesort($2K/3$) is true. Since stoogesort always relies on being able to successfully sort a

subset that is within the scope of our inductive assumption, by the principal of mathematical inducation, stooge sort is true. $\square$

## 2

**6.17 from book:** Given an unlimited supply of coins of denominations $x_1, x_2, \ldots, x_n$, we wish to make change for a value $v$; that is, we wish to find a set of coins whose total value is $v$. This might not be possible: for instance, if the denominations are 5 and 10 then we can make change for 15 but not for 12. Give an $O(nv)$ dynamic-programming algorithm for the following problem. Input: $x_1, \ldots, x_n; v$. Question: Is it possible to make change for $v$ using coins of denominations $x_1, \ldots, x_n$ ?

**Solution:** Make change function:

```
MAKE_CHANGE(A, v):
    coins[$i$] := 0 for 0 to $v$
    for $i$ from 1 to $v$:
        for $j$ from 1 to $A$:
            if $i > j$:
                min := coins[$i - j$] + 1
                if min < coins[$i$]:
                    coins[$i$] := min
    if coins[$i$] != 0:
        return True
    return False
```

## 3

**Problem:** Additional question on 6.17. Show how one can reduce the problem specified by 6.17 into a knapsack problem (with repetition). Reducing to knapsack problem means that for any instance of the coin-change problem specified in 6.17, we can turn it into a knapsack problem, and apply an algorithm for knapsack, and use its solution to decide the solution for the original coin change problem.

**Solution:** The problem in 6.17 is a coin-change problem. The coin change problem can be reduced to a knapsack problem. To do so, let the denominations of coins be the values of items to be placed in the knapsack and let the total value of items be equal to the weight. The weight of an individual element is ignored.

## 4

**6.26 from book:** Sequence alignment. When a new gene is discovered, a standard approach to understanding its function is to look through a database of known genes and find close matches. The closeness of two genes is measured by the extent to which they are aligned. To formalize this, think of a gene as being a long string over an alphabet $\sum = \{A, C, G, T\}$. Consider two genes (strings) $x = ATGCC$ and $y = TACGCA$. An alignment of $x$ and $y$ is a way of matching up these two strings by writing them in columns, for instance:

$$\begin{array}{ccccccc} - & \text{A} & \text{T} & - & \text{G} & \text{C} & \text{C} \\ \text{T} & \text{A} & - & \text{C} & \text{G} & \text{C} & \text{A} \end{array}$$

Here the "-" indicates a "gap." The characters of each string must appear in order, and each column must contain a character from at least one of the strings. The score of an alignment is specified by a scoring matrix $\delta$ of size $(|\sum|+1)*(|\sum|+1)$, where the extra row and column are to accommodate gaps. For instance the preceding alignment has the following score:

$$\delta(-,T)+\delta(A,A)+\delta(T,-)+\delta(-,C)+\delta(G,G)+\delta(C,C)+\delta(C,A)$$

Give a dynamic programming algorithm that takes as input two strings $x[1\ldots n]$ and $y[1\ldots m]$ and a scoring matrix $\delta$, and returns the highest-scoring alignment. The running time should be $O(mn)$.

**Solution:** This solution builds on the dynamic programming algorithm for edit distance. Instead of finding a minimum edit, we find a maximum alignment score.

```
$n$ is the length of $A$
$m$ is the length of $B$
$\delta$ is a scoring function of size $(|\sum|+1)*(|\sum|+1)$
```

```
SEQ_ALIGN($A$, $B$, $\delta$):
    score[$i$, 0] = 0, for $0,...,n$
    score[0, $j$] = 0, for $0,...,m$
    for $i$ from 1 to $n$:
        for $j$ from 1 to $m$:
            ins = score[$i-1$, $j$] + 1
            del = score[$i$, $j-1$] + 1
            sub = score[$i-1$, $j-1$] + $\delta(A_i, B_j)$
            score[$i$, $j$] = $\max\{$ins, del, sub$\}$
    return score[$n$, $m$]
```