# Written Homework #4

Richard Dizon
CS 4102: Algorithms
University of Virginia

October 21, 2016

## 1

For $n$ doors, we know for certain that there are $2^n$ total possible combinations of opened and closed doors. We know for certain that there is only one unique way $S$ for both values $n$ and $n-1$ where all doors are closed and the first $n-1$ doors are closed respectively. Beyond that, we can recursively generate a table of $n$ columns and $n^2$ rows and enumerate each row to be the binary representation of that row number starting from 0. We can analyze each row individually through a separate function to determine the amount of secured doors $S$ knowing again that each row represents a unique combination of open and closed doors. From that function's calculations, we can map out the number of distinct ways the door can be locked to each of those rows and enumerate them to properly find $S$.

## 2

The brute force algorithm would be to consider all values $i$, $j$ and $k$ such that each of their values are less than or equal to the string length $k$ and call the $SQ$ function multiple times. This will have a run time of $O(k*k*k)$, finding every possible value we can generate.

## 3

To be more efficient, we want to consider the actual value of $I$ and try to have as close of a $SQ$ value to that as we can by comparing the the letters at each of the indicies of $A$ and $B$. The three recursive cases would be that either the letters are the same, differ and result in a $SQ$ value less than or equal to $I$ or differ and result in a $SQ$ value greater than $I$. A more efficient approach can reduce this down to $O(k * k)$ calls by instead iterating and deliberately choosing the letters to start with through the recursive function.

## 4

The first step would be to take the entire list of $n$ boxes and sort them into an array $a1$ from lowest volume to highest volume where each value in the array contains a nested array with that individual box's dimensions also in sorted order from least to greatest. From here, for each item in $a1$ we generate a table with dimensions of $nxn$ where each row indicates boxes which it will fit in. This can be determined by comparing each sorted-dimension value to each other; if a column has a digit where it is less than or equal to the row's digit then the box will not fit into the box indicated by the column.

We can now dynamically find out the optimum box set. We construct arrays containing subsets of $a1$, ex. $\{a_n\}, \{a_{n-1}, a_n\}, ..., \{a_0, ..., a_n\}$. We want to find optimal substructure by determining the longest list of boxes a box can be nested in by comparing it to the boxes in its subset. A box can nested in a larger box if its volume is smaller and each individual value in the sorted dimension is less than the counterpart. Optimal substructure will be found for each box size starting from the largest going down to conserve time. Each invocation will try to find this by comparing the first element to the optimal length for every subsequent element in its list and find the largest list of number of boxes.

Set-up itself will cost $O(nlogn)$ given that the boxes will initially be sorted by their volume. Each box will also sort its dimensions, but for a normal three-dimensional box, this will only take at most two swaps to get in order. The bulk of the work done in the second paragraph will take an additional $O(n^2)$ which will overtake the initial run-time value and will therefore run at this speed.

## 5

Given the list of runs $R$ and each of their ski lengths, we can try to generate all of the possible in-order combinations of trips that fall within the length of a day. The first generated combination could consider the greedy solution where you combine as many slopes together as you can until you hit the maximum time spent and store the time wasted on each day. The next generated combination should attempt to shift the value of the last slope of the first day over to the second day and see if it results in a different amount of days spent while also falling within the more profitable range of time resulting in the post-ski break. We do this in multiple instances for each day, shifting the tail-end slope over to the next day, still storing the time-wasted for each combination.

For any combination with the same minimum amount of days, the total time wasted across the days should be similar, but what differs is the value generated from the $twd(t)$ function. $t$-values lying within the range of 1 to $m$ are highly valued where values exceeding $m$ become more costly. Another function can calculate the optimized value from the combinations of the smallest number of days.

## 6

As it turns out, the dynamic algorithm to solve this ends up being similar to the Fibonacci sequence algorithm where the floor legal input value is 1 with a run time of $O(2^n)$.

```
def unique_board(dominoes):
 if dominoes <= 0:
  return 0
 elif dominoes == 1:
  return 1
 else:
  return unique_board(dominoes-1)+unique_board(dominoes-2)
```

## 7

The only difference this time is that in the recursive step, each new call of the function is multiplied by a factor of five with a similar runtime of $O(2^n)$, as follows:

```
def unique_board(dominoes):
 if dominoes <= 0:
  return 0
 elif dominoes == 1:
  return 1
 else:
  return 5*unique_board(dominoes-1)+5*unique_board(dominoes-2)
```