1 Proofs of Correctness

Good things to consider when writing proofs:

- First state what you are going to prove. What does it mean for your algorithm to be correct? For every sentence in your proof, make sure it can relate back to this. Then state how you are going to prove this. Give the reader something to expect.

- Consider an arbitrary input, case, etc. Use variables for any arbitrary values of the input (ex: an arbitrary list of size $n$). Unless we ask for it, a proof by example is not adequate.

- Employ the appropriate proof technique (induction, contradiction, cases, etc.)

- Don’t repeat yourself if possible. Focus on the crucial parts of the proof. Short and concise proofs are more powerful than long, wordy proofs.

- When using pseudocode, reference specific lines in your proof.

- Don’t forget a conclusion! It should match what you said you would prove.

1.1 Selection Sort

Prove the correctness of the Selection Sort algorithm described below.

```plaintext
Selection Sort(L)
1  n = length(L)
2  if n ≤ 1
3    halt
4  x =\text{min}(L)
5  swap x and L[0]
6  Selection Sort(L[1 : n])
```

The proof of correctness will look something like the following:
• **State what you are going to prove and how you are going to prove it:** Given a nonempty input array of integers, we will prove by induction that selection sort will correctly sort the input array from smallest to largest.

• **Consider an arbitrary input, case, etc.** Consider an arbitrary nonempty input array of \( n \) integers.

• **Employ the appropriate proof technique - in this case, induction:** If \( n \) is 1, then the input list is sorted, the condition on line 2 of the algorithm is satisfied, and the program will halt. Therefore, the array is unchanged and selection sort correctly sorted the array. Assume selection sort correctly sorts any arbitrary input array of integer size \( k \), where \( k > 1 \). Consider now applying selection sort to an arbitrary nonempty array of length \( k + 1 \). In the initial call to the selection sort algorithm, line 5 swaps the first element of this array with the minimum element in the entire array. Now the first element of the array is correctly sorted. Then line 6 of the algorithm calls selection sort on the remaining unsorted \( k \) elements. By our induction hypothesis, we can now successfully apply selection sort to sort the remaining \( k \) elements in the array. Thus we can correctly sort an array of \( k + 1 \) integers.

• **Don’t repeat yourself if possible:** Check: did we repeat ourselves?

• **Conclusion:** We conclude by induction that selection sort is correct.

2 **Dynamic Programming**

Dynamic programming is a way to efficiently solve complex problems by cleverly building on solutions to smaller sub-problems. Not every problem you encounter can be solved with dynamic programming, but you will find that the problems that can be solved with dynamic programming have essentially the same structure. This makes it easy to develop a consistent strategy for solving dynamic programming problems - and more importantly, proving that your strategy is correct.

2.1 **Dynamic Programming Proofs**

Dynamic programming proofs are an extension off of our basic proof framework.

• **First state what you are going to prove and how you are going to prove it.** With dynamic programming this is usually induction over filling out each element of your table. Specify what “correct” means and also the order in which you will fill out the table.

• **Employ the appropriate proof technique.** With dynamic programming this is usually induction.
  
  – **Base case:** Explain correctness of the initialization of the table.
Consider an arbitrary input. For dynamic programming this is an entry into the table. “Consider an arbitrary entry into table, $T, T(i,j,k...)$."

- **Inductive Hypothesis:** Assume all entries filled out before $T(i,j...)$ are correct.
- Consider the optimal solution and the last decision made to get to this solution.
- **Break down the possible values of $T(i,j..)$ into cases.** Then explain why in each case your algorithm can do at least as well as the optimal solution.
- **Verify that any previous entries used to compute $T(i,j,k..)$ are computed before $T(i,j,k...)$ is.

- Don’t repeat yourself if possible.
- Don’t forget a conclusion!

### 2.2 General Doubleday

You are a decorated general in charge of $n$ soldiers. You can deploy any number of these soldiers across $m$ possible battlefields occupied by various enemy troops. Having scouted ahead (you are a decorated general, after all), you have a formula $f(i,j)$ that tells you how many of your soldiers will survive deployment if you deploy $j$ soldiers to the $i^{th}$ battlefield. Assuming that you must deploy all of your troops, devise an algorithm for determining the maximum possible number of survivors. Prove the runtime of your algorithm. Prove that your algorithm is correct.

### 2.3 Winning the war

**Algorithm**

Your strategy is to create a $(m+1) \times (n+1)$ (battlefields $\times$ soldiers) table $T$. Indexing from zero, position $(i,j)$ represents the maximum number of survivors attainable by allocating $j$ troops over the first $i$ battlefields. In the base case, we initialize $T(1,j) = f(1,j)$ for $0 \leq j \leq n$ (note that $T(0,j)$ is ignored for all $0 \leq j \leq n$, as there is at least 1 battlefield). Then we traverse $T$ row by row (where $i$ represents rows) and for a given square $T(i,j)$, set:

$$T(i,j) = \max_{0 \leq k \leq j} T(i-1,j-k) + f(i,k)$$

The maximum possible number of survivors is then finally determined from $T(m,n)$.

**Proof of Correctness**

We will show by induction on $i$ and $j$, where $1 \leq i \leq m$ and $0 \leq j \leq n$ and
traversed row by row, that our algorithm correctly computes the maximum possible number of survivors when deploying \( j \) soldiers across the first \( i \) battlefields. As a base case for the induction, we note that \( T(1, j) \) is correctly filled with the number of survivors attained by deploying all \( j \) soldiers onto the first battlefield, for all \( 0 \leq j \leq n \).

Now consider for an arbitrary assortment of \( m \) battlefields, \( n \) soldiers, and an arbitrary survivor function \( f \), arbitrary positive integers \( i, j \). Assume that all entries filled before \( T(i, j) \) in the table are correct. Consider the maximum possible number of survivors, and consider the last decision made to reach this optimal outcome, namely how many soldiers to allocate to battlefield \( i \), the “newest” battlefield. We can allocate anywhere from 0 to \( j \) soldiers to the \( ith \) battlefield. Let this number of soldiers be \( k \). Therefore the optimal solution represents the maximum value of the number of survivors on battlefield \( i \) if we allocate it \( k \) soldiers, plus the maximum number of survivors across battlefields 0 through \( i - 1 \) given a total deployment of \( j - k \) soldiers, over all \( k \) \((0 \leq k \leq j)\). The former quantity is represented by \( f(i, k) \) while the latter, by our inductive hypothesis, is represented by \( T(i - 1, j - k) \). Thus for this arbitrary \( i \) and \( j \), the optimal solution computes the maximum possible number of survivors as:

\[
\max_{0 \leq k \leq j} T(i - 1, j - k) + f(i, k)
\]

Since our algorithm processes entries row by row, by our induction hypothesis, entries \( T(i - 1, k - j) \) for all \( 0 \leq k \leq j \), must have been correctly filled before \( T(i, j) \). As our algorithm represents the maximum possible number of survivors with \( T(i, j) \), and also computes it as

\[
T(i, j) = \max_{0 \leq k \leq j} T(i - 1, j - k) + f(i, k)
\]

it follows that our algorithm exactly captures the optimal outcome. We thus conclude our inductive proof that our algorithm correctly fills out a table of maximum possible survivors and therefore correctly computes the best strategy for deploying \( n \) soldiers across \( m \) battlefields as \( T(m, n) \).

**Runtime Analysis**

The runtime of this algorithm is \( O(mn^2) \). For each entry in \( T \), we take a max over \( O(n) \) sums. The size of \( T \) is \( O(mn) \). Retrieving the final answer is done by reading from the table in \( O(1) \). Since creating and populating the table runs in \( O(mn^2) \) while retrieving the answer runs in \( O(1) \), the final runtime is \( O(mn^2) \).