# Algorithms Assignment 10
# Fall 2022

Daniel Grant

10 November 2022

## Question 1

**Finding the singleton:** You are given a sorted array of numbers where every value except one appears exactly twice, and one value appears only once. Design an efficient algorithm for finding which value appears only once.

Note: A general solution should not assume anything about the numbers in the array; specifically, they may not be in a small range, and may not be consecutive.

**Example:** Here are some example inputs to the problem:

$$1, 1, 2, 2, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8$$

$$10, 10, 17, 17, 18, 18, 19, 19, 21, 21, 23$$

$$1, 3, 3, 5, 5, 7, 7, 8, 8, 9, 9, 10, 10$$

*We expect: pseudocode, a clear (and brief) English description of what the algorithm is doing and why it is correct; running time analysis.*

1. **Pseudocode**

**Algorithm 1:** Finding the Singleton

**input :** A sorted array of numbers, $A$, where every value except one appears exactly twice, and one value appears only once.
**input :** The length of the array, $n$.
**output:** The value that appears only once.

```
1  Function FindSingleton(A, n):
2      l ← 0                                                    // left index
3      r ← n − 1                                                // right index
4      while l < r do
5          m ← ⌊(l+r)/2⌋                                        // middle index
6          if A[m] = A[m − 1] then
                                  // if the middle value is equal to the value before it
7              if m mod 2 = 0 then
                                                                // if the middle index is even
8                  r ← m − 2
9              else
                                                                // if the middle index is odd
10                 l ← m + 1
11         if A[m] = A[m + 1] then
                                  // if the middle value is equal to the value after it
12             if m mod 2 = 0 then
                                                                // if the middle index is even
13                 l ← m + 2
14             else
                                                                // if the middle index is odd
15                 r ← m − 1
16     return A[m]                                              // return the value at the middle index
```

2. **Description**

The idea behind this code is to use a binary-search algorithm to iteratively check a given position, and see whether the element to the right of left of it is equal to that given element.

If the element to the left of the given element is equal to it, then we know that the element is not the singleton, and we can eliminate half of the array. If the element to the right of the given element is equal to it, then we know that the element is not the singleton, and we can eliminate half of the array. If neither of these are true, then we know that the element is the singleton, and we can return it.

However, we need to be careful about the indices that we are checking. If the current middle element of the sub array we are looking at is at an odd index, then one of two cases can occur:

(a) The element to the left of the middle element is equal to the middle element. In this case, we know that the singleton is to the right of the middle element, and we can eliminate the left half of the array.

(b) The element to the right of the middle element is equal to the middle element. In this case, we know that the singleton is to the left of the middle element, and we can eliminate the right half of the array.

If the current middle element of the sub array we are looking at is at an even index, then one of two cases can occur:

(a) The element to the left of the middle element is equal to the middle element. In this case, we know that the singleton is to the left of the middle element, and we can eliminate the right half of the array.

(b) The element to the right of the middle element is equal to the middle element. In this case, we know that the singleton is to the right of the middle element, and we can eliminate the left half of the array.

3. **Running Time Analysis**

Since this algorithm is implementing a standard binary search algorithm, the running time is $O(\log n)$.

**Note:-**

Just like in a traditional binary search, this algorithm is also computing a constant-time operation for each iteration, so there is no other complexities to consider. Furthermore, as this is an iterative implementation, there will also be no recursive overhead to deal with.

## Question 2

**Art gallery guarding:** In the art gallery guarding problem we are given a line $L$ that represents a long hallway in an art gallery. We are also given a set $X = \{x_0, x_1, x_2, \ldots, x_{n-1}\}$ of real numbers that specify the positions pf paintings in this hallway; assume that each painting is a point. Suppose that a single guard can protect all the paintings within a distance at most 1 of his or her position, on both sides.

Design an algorithm for finding a placement of guards that uses the minimum number of guards to guard all the paintings in $X$. You can assume that the positions of the paintings are sorted. Briefly argue why your algorithm is correct and analyze its running time.

*We expect: The algorithm as pseudocode, a brief explanation, analysis and justification of correctness i.e. why does it always assure the minimum number of guards? Remember that to show that a greedy algorithms is correct it is sufficient to show that there exists an optimal solution that includes the first greedy choice.)*

1. **Pseudocode**

**Algorithm 2:** Finding the Optimal Guard Placement

**input :** A sorted array of numbers, $X$, where each number represents the position of a painting.
**input :** The length of the array, $n$.
**output:** The number of guards required to guard all the paintings.

```
1  Function ArtGallery(X, n):
2      guardCount ← 1                                   // number of guards
3      i ← 1                                            // index of current painting
4      curGuard ← X[0] + 1                              // position of current guard
5      while i < n do
6          if X[i] > curGuard + 1 then
                                  // if the current painting is not within the range of the current guard
7              curGuard = X[i] + 1
8              guardCount ← guardCount + 1
9          i ← i + 1                                    // increment the index of the current painting
10     return guardCount
```

2. **Description**

The idea behind this code is to use a greedy algorithm to iteratively check the position of the current guard, and see whether the next painting is within the guard's range. If the next painting is within the guard's range, then we know that the guard can protect that painting, and we can move on to the next painting. If the next painting is not within the guard's range, then we know that the guard cannot protect that painting, and we need to place a new guard at that painting's position.

We can initialize the first guard as being at the position of the first painting, plus one. This is because we know that if the guard is one unit away from that first painting, they are as far away as possible to that painting while still being able to protect it.

We can then iterate through the array of paintings, and check whether the next painting is within the guard's range. If it is, then we can move on to the next painting. If it is not, then we know that we need to place a new guard at that painting's position. We can then increment the number of guards, and move on to the next painting.

3. **Running Time Analysis**

Since this algorithm is implementing a standard linear search algorithm, the running time is $O(n)$.

**Note:-**

Just like in a traditional linear search, this algorithm is also computing a constant-time operation for each iteration, so there is no other complexities to consider. Furthermore, as this is an iterative implementation, there will also be no recursive overhead to deal with.

4. **Justification of Correctness**

   (a) **Optimal Substructure**

   The optimal substructure property states that an optimal solution to a problem can be constructed from optimal solutions to its subproblems. In this case, the subproblem is the problem of finding the optimal guard placement for a subarray of the original array.

   (b) **Greedy Choice**

   The greedy choice property states that at each step of the algorithm, we make a choice that looks best at the moment, and we hope that this choice will lead to an optimal solution. In this case, the greedy choice is to place a guard at the position of the next painting that is not within the range of the current guard, plus one.

   (c) **Correctness**

   The correctness of this algorithm can be determined by showing that the greedy choice is always optimal. This can be done by showing that the greedy choice is always a part of the optimal solution.

   The greedy choice is always a part of the optimal solution because it is always the case that the optimal solution will include the first painting that is not within the range of the current guard. This is because if the optimal solution does not include the first painting that is not within the range of the current guard, then the optimal solution will include the first painting that is not within the range of the current guard, and the first painting that is not within the range of the current guard will be the next painting that is not within the range of the current guard.

   Therefore, the greedy choice is always a part of the optimal solution, and the greedy choice will be optimal.

   😄

## Question 3

**Unbounded knapsack:** We have $n$ items, each with a value and a positive weight; Item $i$ has value $v_i$ and weight $w_i$. We have a knapsack that holds maximum weight $W$. In this problem you'll consider a variation of the $0-1$ Knapsack problem, where we have infinite copies of each item. Describe an algorithm that, given $W, \{w_1, \ldots, w_n\}$ and $\{v_1, \ldots, v_n\}$, finds the maximal value that can be loaded into the knapsack.

1. A friend proposes the following greedy algorithm: start with the item with the largest cost-per-pound, and pick as many copies as fit in the backpack. Repeat. Show your friend this is not correct by coming up with a counter-example.

   *We expect: a small example, the greedy solution, and the optimal solution*

2. Define your sub-problem and state clearly what its argument(s) represent and what it returns.

   Hint: With the classical knapsack, once you take an item, you cannot take it again, so you have fewer items to choose from. So the subproblem we use has to keep track of both the knapsack size as well as which items are allowed in the knapsack. We used *knapsack* $(w, i)$ to be the maximum value we can get for a knapsack of weight $w$ considering items 1 through $i$. For the unbounded knapsack, we have infinite supplies of each item. So we don't need to keep track of which items we already included (only of the size of the knapsack).

3. Argue optimal sub-structure: Hint: Suppose the optimal solution contains item $i$. Then the remaining part of the optimal solution must be an optimal solution for .... .

4. Come up with a recursive definition for the subproblem you chose in (2) Hint: Consider all the choices, and pick the best.

5. Develop a DP solution - either top-down or bottom up. What is the running time of your algorithm?

1. **Greedy Counter-Example**

**Example 0.1** (Unbounded KnapSack Example)

In the case of the table below, if you had a knapsack of size 4 the greedy algorithm would choose to take item 3, since it has the highest value per weight ($\frac{5}{3} > \frac{3}{2} > \frac{4}{4}$).

| Item | Value | Weight |
|---|---|---|
| 1 | 4 | 4 |
| 2 | 3 | 2 |
| 3 | 5 | 3 |

.

However, the optimal solution would be to take item 2, since although it does not have the highest value per weight ratio, you could fit two of them into the knapsack, totaling to a value of 6 as compared to the knapsack of value 5 for the greedy solution.

→ *General Case:*

In general, this will not work because the greedy algorithm will not always choose the optimal solution. This is because the greedy algorithm will always choose the item with the highest value per weight ratio, but this does not always mean that it will be the optimal solution.

**Algorithm 3:** Greedy Unbounded Knapsack Algorithm

**input :** A list of $n$ items, each with a value $v_i$, and a weight $w_i$, and a knapsack that holds a maximum weight of $W$.

**output:** The maximum value that can be loaded into the knapsack.

```
1  Function GreedyKnapsack(W, {w_1,...,w_n}, {v_1,...,v_n}, n):
2      V = 0                                              // Initialize the value of the knapsack to 0.
3      while W > 0 do
4          i = FindBestValue({w_1,...,w_n}, {v_1,...,v_n}, n, W)   // Find highest value ratio that fits.
5          W = W − w_i                                    // Remove the weight of the item from the knapsack.
6          V = V + v_i                                    // Add the value of the item to the knapsack.
7      return V
8  Function FindBestValue({w_1,...,w_n}, {v_1,...,v_n}, n, maxWeight):
9      bestIndex ← 0                                      // Initialize the best index to 0.
10     bestValue = v_0/w_0                                // Initialize the best value to the value of the first item.
11     for i ← 1 to n do
12         if w_i ⩽ maxWeight then
13             value ← v_i/w_i                            // Calculate the value of the item.
14             if value > bestValue then
15                 bestValue ← value                      // Update the best value.
16                 bestIndex ← i
17     return bestIndex                                   // Return the index of the best item.
```

→ *Algorithm Description:*

The algorithm starts by initializing the value of the knapsack to 0. Then, it loops through the items, finding the item with the highest value per weight ratio that fits in the knapsack. It then removes the weight of the item from the knapsack and adds the value of the item to the knapsack. It then repeats this process until the knapsack is full.

However, as discussed above, this will not be optimal. The optimal solution will be to take item 2, since although it does not have the highest value per weight ratio, you could fit two of them into the knapsack, totaling to a value of 6 as compared to the knapsack of value 5 for the greedy solution. This could be implemented in a very similar manner to the traditional bounded 0-1 Knapsack problem.

In essence, instead of deciding to include an item or not, we will decide to include an item $k$ times, where $k$ is the maximum number of times that the item can fit in the knapsack. As I will show in the following sections, this approach will yield the optimal solution.

2. **Subproblem Definition**

The subproblem that we will be using is the following:

$$\text{UnboundedKnapSack}(W, \{w_1, \ldots, w_n\}, \{v_1, \ldots, v_n\})$$
$$= \max_{i \in \{1,\ldots,n\}} \{\text{UnboundedKnapSack}(W - w_i, \{w_1, \ldots, w_n\}, \{v_1, \ldots, v_n\}) + v_i\}$$

**Note:-**

The argument for this subproblem is the weight of the knapsack, and the set of weights and values of the items. The return value is the maximum value that can be loaded into the knapsack.

3. **Optimal Substructure**

The optimal substructure of this problem can be explained in the following manner:

Suppose you have an optimal solution, with containing some item $i$ $k$ times. Then, the optimal solution for the subproblem of the knapsack with a weight of $W - kw_i$ will be the optimal solution for the knapsack with a weight of $W$. This is because the optimal solution for the knapsack with a weight of $W$ will be the optimal solution for the knapsack with a weight of $W - kw_i$ plus the value of the item $i$ $k$ times.

4. **Recursive Definition**

The recursive definition of non-dynamic recursive algorithm is the following (*Using acronym UKS to represent UnboundedKnapSack function call, for brevity*):

$$\begin{aligned}
\text{UKS}(W, \{w_1, \ldots, w_n\}, \{v_1, \ldots, v_n\}) &= \max_{i \in \{1,\ldots,n\}} \{\text{UKS}(W - w_i, \{w_1, \ldots, w_n\}, \{v_1, \ldots, v_n\}) + v_i\} \\
&= \max_{i \in \{1,\ldots,n\}} \{\text{UKS}(W - w_i, \{w_1, \ldots, w_n\}, \{v_1, \ldots, v_n\}) + v_i\} \text{ if } W - w_i \geqslant 0 \\
&= 0 \text{ if } W - w_i < 0
\end{aligned}$$

The first line is the recursive definition of the problem. The second line is the recursive definition of the problem with the base case. The third line is the base case of the problem.

To represent this in a more formal manner, we can use the following recurrence relation:

$$T(n) = T(n, W - w_n) + T(n-1, W) + 1$$

This relation has a base case when $W \leqslant 0$ or $n = 0$:

$$T(n) = \begin{cases} 0 & \text{if } W \leqslant 0 \text{ or } n = 0 \\ T(n, W - w_n) + T(n-1, W) + 1 & \text{otherwise} \end{cases}$$

Simplifying this relation, we get the following:

$$\begin{aligned}
T(n) &= 2^{min(W+n, n+n)} \\
&= 2^{min(W,n)+n}
\end{aligned}$$

Notice that for this solution, the time complexity is exponential. This is because the number of subproblems is exponential in the number of items. In the worst case, there could be $2^{2n}$ function calls in the case of there being more items than capacity in the knapsack. In the second case, where $W < n$, there could be $2^W$ function calls, then an additional $2^n$ in the worst case (generalizes to simply $2^{W+n}$ calls).

**Algorithm 4:** Finding the optimal solution to the unbounded Knapsack problem without dynamic programming.

**input :** The weight of the knapsack, $W$, and the set of weights and values of the items, $\{w_1, \ldots, w_n\}$ and $\{v_1, \ldots, v_n\}$.
**output:** The maximum value that can be loaded into the knapsack.

```
Function NonDPUnboundedKnapsack(W, {w_1, ..., w_n}, {v_1, ..., v_n}):
    if W ⩽ 0 or n = 0 then
        return 0                                                    // Base case
    return max(NonDPUnboundedKnapsack(W − w_n, {w_1, ..., w_n}, {v_1, ..., v_n}) + v_n,
     NonDPUnboundedKnapsack(W, {w_1, ..., w_{n−1}}, {v_1, ..., v_{n−1}}))
```

→ *Algorithm Description:*

This algorithm is a recursive algorithm that finds the optimal solution to the unbounded 0-1 Knapsack problem without dynamic programming. The algorithm takes in the weight of the knapsack, $W$, and the set of weights and values of the items, $\{w_1, \ldots, w_n\}$ and $\{v_1, \ldots, v_n\}$. The algorithm returns the maximum value that can be loaded into the knapsack.

**Note:-**

For the dynamic implementation of this problem, which is very similar to this implementation, we will also be using a 2D array to store the values of the subproblems. The first dimension will be the number of items, and the second dimension will be the weight of the knapsack. This will allow us to store the values of the subproblems in a bottom-up manner, and then retrieve the optimal solution from the top of the array.

The runtime of this algorithm will be $O(n \cdot W)$, where $n$ is the number of items and $W$ is the weight of the knapsack.

This is because we will be looping through the items, and then looping through the weights of the knapsack, which produces the $O(n \cdot W)$ runtime.

5. **Dynamic Programming Solution**

**Algorithm 5:** Finding the optimal solution to the unbounded 0-1 Knapsack problem with dynamic programming.

**input :** The weight of the knapsack, $W$, and the set of weights and values of the items, $\{w_1, \ldots, w_n\}$ and $\{v_1, \ldots, v_n\}$.
**input :** A table of values, $T$, that stores the values of the subproblems. (This table is of size $W \times n$)
// Initialized with all −1s
**output:** The maximum value that can be loaded into the knapsack.

```
Function DPUnboundedKnapsack(W, {w_1, ..., w_n}, {v_1, ..., v_n}, table):
    if T[W][n] ≠ −1 then
        return T[W][n]          // If the value of the subproblem has already been calculated, return it.
    if W ⩽ 0 or n = 0 then
                // If the weight of the knapsack is less than or equal to 0, or there are no items, return 0.
        return 0
    T[n, W] = max(NonDPUnboundedKnapsack(W − w_n, {w_1, ..., w_n}, {v_1, ..., v_n}) + v_n,
     NonDPUnboundedKnapsack(W, {w_1, ..., w_{n−1}}, {v_1, ..., v_{n−1}}))
    return T[n, W]                                  // Return the value of the subproblem.
```

→ *Algorithm Description:*

This algorithm is identical in terms of functionality to the non-dynamic programming solution, but it uses a table to store the values of the subproblems. The algorithm takes in the weight of the knapsack, $W$, and the set of weights and values of the items, $\{w_1, \ldots, w_n\}$ and $\{v_1, \ldots, v_n\}$. The algorithm also takes in a table of values, $T$, that stores the values of the subproblems. (This table is of size $W \times n$). The algorithm similarly returns the maximum value that can be loaded into the knapsack. The runtime does differ, however. This is explained in the note in the **Recursive Definition**[4] section above.

😄