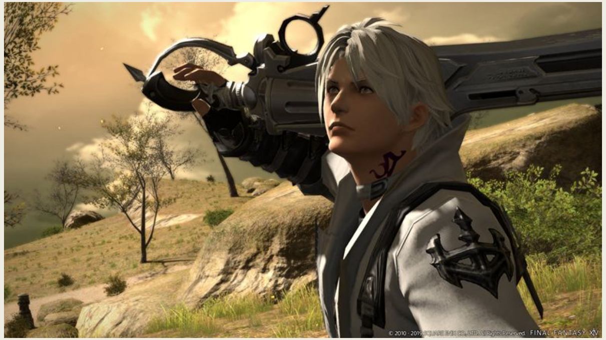
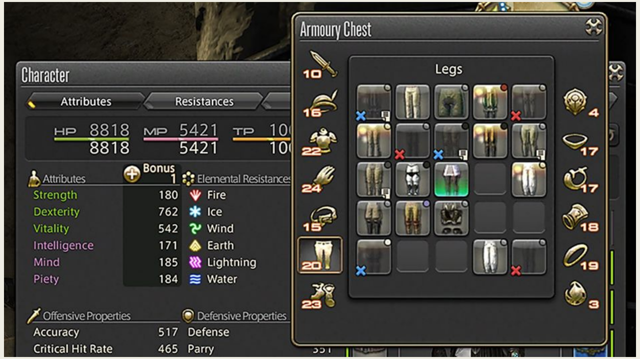
Inventory Management

Thanh Nguyen and Karl Wise

Initial Concept: Inventory Management

- Inspired by Final Fantasy XIV
- Rank the items collected in the game
- Optimize items in relation to bag space
- QuickSort
- TimSort
- Bubble Sort
- Bucket Sort





QuickSort

- Takes a pivot point and iterates through a given array round said pivot.
- Multiple ways to choose a pivot point:
  - *The first element*
  - *The last element*
  - *Any random element*
  - *The median element*
- Runs in O(n^2) run time

```
public void quickSort(int arr[], int start, int end){
    if(start < end){
        int partitionIndex = partition(arr, start, end);

        quickSort(arr, start, partitionIndex - 1);
        quickSort(arr, partitionIndex + 1, end);
    }
}

private int partition(int arr[], int start, int end){
    int pivot = arr[end];
    int i = (start - 1);

    for(int j = begin; j < end; j++){
        if(arr[j] <= pivot){
            i++;

            int swapTemp = arr[i];
            arr[i] = arr[j];
            arr[j] = swapTemp;
        }
    }

    int swapTemp = arr[i + 1];
    arr[i + 1] = arr[end];
    arr[end] = swapTemp;

    return i+1;
}
```

TimSort

- Native Sorting Algorithm to Java and Python
  - *Java: arrays.sort()*
  - *Python: sorted() and sort()*
- Combination of Insertion Sort and Merge Sort
  - *Sorts small pieces with Insertion Sort*
  - *Merges the pieces with Merge Sort*
- Run in O(n log n) time

```java
//Example Arrays.sort() usage
import.java.util.Arrays;
public static void main(String[] args){
    int[] arr = {15, 5, 8, 3, 7, 6};
    Arrays.sort(arr);
    System.out.println("Here is your sorted array: %s", Arrays.toString(arr));
}
```

Bubble Sort

- Works by repeatedly swapping adjacent elements until they are in the right order

- Runs in O(n^2) run time

- Has no real world use, primarily used as an education tool.

```
public static void bubbleSort(int arr[]){
    int n = arr.length;
    for(int i = 0; i < n - 1; i++)
        for(int j = 0; j < n - 1; j++)
            if(arr[j] > arr[j + 1]){
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
}
```

BucketSort

- Distributes elements of an array into “buckets”
  - *Each bucket is sorted using a different sorting algorithm or recursively using bucket sort.*
- Runs in O(n^2) run time

```java
import java.util.*;
public static void bucketSort(float arr[], int n){
    if(n <= 0)
        return;
    //Override the Java rules on generics manipulation
    @SuppressWarngs("unchecked")
    Vector<Float>[] buckets = new Vector[n];

    for(int i = 0; i < n; i++){
        buckets[i] = new Vector<Float>();
    }

    for(int i = 0; i < n; i++){
        float idx = arr[i] * n;
        buckets[(int)idx].add(arr[i]);
    }

    for(int i = 0; i < n; i++){
        Collections.sort(buckets[i]);
    }

    int index = 0;
    for(int i = 0; i < n; i++){
        for(int j = 0; j < buckets[i].size(); j++){
            arr[index++] = buckets[i],get(j);
        }
    }
```

The 0-1 KnapSack Problem (KSP)

- A problem in computation optimization
  - *Items with a weight and value are compared within an array.*
  - *Selects the item set that would provide the most value, while efficiently handling the weight of the bag.*

- Simplest implementations of KSP are solved through brute force, but run extremely slow O(2^n)

- Can be optimized using a programming technique called Dynamic Programming

Dynamic Programming

- Storing the results of subproblems in a recursive manner, so that we don’t have to recompute them later.

- Two ways to create Dynamic Programming Structure
  - *Top Down:*
    - Memoization
  - *Bottom-Up:*
    - Tabulation

Dynamic Programming: Memoization

- Formulate through problem recursively using the solution of its sub-problems
- Stores the solution of every sub-problem into an array
- Whenever a duplicate sub-problem is called, references the solution within the array, instead of re-computing

Dynamic Programming: Tabulation

- Starts from the solving the lowest level sub-problem to help solve the next level subproblem.
- Solves all the smaller subproblems needed to solve the bigger subproblems.
- Solves all the subproblems iteratively until all subproblems are solved
- Doesn’t throw a stack overflow error but harder to design

Knapsack Problem: Brute Force Method

- Evaluates all subsets of items to find the total weight and value of every subset.
  - *Only considers the subsets whose weight is less than the max weight of the bag.*
  - *The subset with the maximum value and with a weight less than or equal to the amount available in the bag.*
- Time Complexity: O(2^n)
  - *Has redundant subproblems*
- Auxiliary Space: O(1)
  - *Doesn't use extra data structure to store values*

```java
import java.time.*;

public class KnapBrute{
    static int max(int a, int b){
        return (a > b) ? a : b;
    }

    static int knapSack(int W, int wt[], int val[], int n){

        if(n == 0 || W == 0)
            return 0;

        if(wt[n - 1] > W)
            return knapSack(W, wt, val, n - 1);

        else
            return max(val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1), knapSack(W, wt, val, n - 1));
    }

    Run | Debug
    public static void main(String[] args){
        Instant start = Instant.now();

        int val[] = new int[] {60, 100, 120, 61, 51, 56, 89, 79, 69, 23, 45};
        int wt[] = new int[] {10, 20, 30, 15, 25, 20, 15, 50, 25, 35, 19 };
        int W = 150;
        int n = val.length;
        System.out.println(knapSack(W, wt, val, n));
        Instant end = Instant.now();

        Duration elapsedTime = Duration.between(start, end);
        System.out.println("The Elapsed Time is: " + elapsedTime);
    }
}
```

Knapsack Problem: Memoization Technique

- Recursive technique to save the result of subproblems to be referenced at later use

- Time Complexity: O(N*W)
  - *Avoids redundant calculated states*

- Auxiliary Space: O(N*W)
  - *Using 2D array data structure to store intermediate states*

```java
public class KnapMemo{

    static int  max(int a, int b){
        return (a > b) ? a : b;
    }

    static int knapSackRec(int W, int wt[], int val[], int n, int dp[][]){
        if(n == 0 || W == 0)
            return 0;

        if(dp[n][W] != -1)
            return dp[n][W];

        if(wt[n - 1] > W)
            return dp[n][W] = knapSackRec(W, wt, val, n - 1, dp);

        else
            return dp[n][W] = max((val[n - 1] + knapSackRec(W - wt[n - 1], wt, val, n - 1, dp)), knapSackRec(W, wt, val, n - 1, dp));

    }

    static int knapSack(int W, int wt[], int val[], int n){
        int dp[][] = new int[n + 1][W + 1];

        for(int i = 0; i < n + 1; i++)
            for(int j = 0; j < W + 1; j++)
                dp[i][j] = -1;

        return knapSackRec(W, wt, val, n, dp);
    }

    public static void main(String[] args){
        int val[] = {60, 100, 120};
        int wt[] = {10, 20, 30};
        int W = 50;
        int N = val.length;

        System.out.println(knapSack(W, wt, val, N));

    }
}
```

Knapsack Problem: Tabulation

- Re-computation of redundant subproblems are avoided by constructing a temporary array using the bottom up approach
- Time Complexity: O(N*W)
  - *N = number of weight element*
  - *W = capacity*
- Auxiliary Space: O(N*W)
  - *Using 2D array*

```java
import java.time.*;

class KnapsackTab1 {

    static int max(int a, int b) {
        return (a > b) ? a : b;
    }

    static void knapSack(int W, int wt[], int val[], int n) {
        int i, w;
        int K[][] = new int[n+1][W+1];

        for (i=0; i <= n; i++) {
            for (w = 0; w <= W; w++) {
                if (i == 0 || w == 0)
                    K[i][w] = 0;
                else if (wt[i-1] <= w)
                    K[i][w] = Math.max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
                else
                    K[i][w] = K[i-1][w];
            }
        }
        int res = K[n][W];
        System.out.println(res);

        w = W;
        for (i=n; i > 0 && res > 0; i--) {
            if (res == K[i-1][w])
                continue;
            else {
                System.out.print(wt[i-1] + " ");
                res = res - val[i-1];
                w = w - wt[i-1];
            }
        }
    }

    public static void main(String args[]) {
        Instant start = Instant.now();

        int val[] = new int[] {40, 100, 50, 60};
        int wt[] = new int[] {20, 10, 40, 30};
        int W = 60;
        int n = val.length;

        knapSack(W, wt, val, n);
        Instant end = Instant.now();
        Duration elapsedTime = Duration.between(start, end);
        System.out.println("\nElapsed Time = " + elapsedTime);
    }
}
```

Knapsack Problem: Tabulation

- Re-computation of redundant subproblems are avoided by constructing a temporary array using the bottom up approach
- Time Complexity: O(N*W)
  - *N = number of weight element*
  - *W = capacity*
- Auxiliary Space: O(2*W)
  - *Using 2D array with only 2 rows*

```java
import java.util.*;
import java.time.*;

class KnapsackTab2 {

    static void knapSack(int W, int wt[], int val[], int n) {
        int i, w;
        int K[][] = new int[2][W+1];

        for (i=0; i <= n; i++) {
            for (w=0; w <= W; w++) {
                if (i == 0 || w == 0)
                    K[i % 2][w] = 0;
                else if (wt[i-1] <= w)
                    K[i % 2][w] = Math.max(val[i-1] + K[(i-1) % 2][w - wt[i-1]], K[(i-1) % 2][w]);
                else
                    K[i % 2][w] = K[(i-1) % 2][w];
            }
        }
        int res = K[n % 2][W];
        System.out.println(res);

        w = W;
        for (i=n; i > 0 && res > 0; i--) {
            if (res == K[(i-1) % 2][w])
                continue;
            else {
                System.out.print(wt[(i-1) % 2] + " ");
                res = res - val[(i-1) % 2];
                w = w - wt[(i-1) % 2];
            }
        }
    }

    Run | Debug
    public static void main(String[] args) {
        Instant start = Instant.now();

        int val[] = {40, 100, 50, 60};
        int wt[] = {20, 10, 40, 30};
        int W = 60;
        int n = val.length;

        knapSack(W, wt, val, n);
        Instant end = Instant.now();
        Duration elapsedTime = Duration.between(start, end);
        System.out.println("\nElapsed Time = " + elapsedTime);
    }
}
```

Lets run some code!

Challenges in Design and Implementation

- Attempting to implement various sorting algorithms within the Knapsack Problem
- Designing the proper logic to output, the items placed in the knapsack
- Understanding the differences between Memoization and Tabulation
- Getting stuck on a more optimal way to implement Memoization and Tabulation
  - *KSP Auxiliary Space: O(2*W)*
    - Using a 2D array but with 2 rows
  - *KSP Auxiliary Space: O(W)*
    - Using 1D Array

Sources

- Knapsack Problem
  - https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
  - https://www.youtube.com/watch?v=nLmhmB6NzcM
- Dynamic Programming
  - https://www.geeksforgeeks.org/dynamic-programming/
  - https://en.wikipedia.org/wiki/Dynamic_programming
- Memoization
  - https://www.geeksforgeeks.org/memoization-1d-2d-and-3d/
  - https://www.javatpoint.com/tabulation-vs-memoization
- Tabulation
  - https://www.javatpoint.com/tabulation-vs-memoization