

Vellore Institute of Technology (VIT)

Design and Analysis of Algorithms (CBS3003) - ETH

Class Number: VL2024250103268

Slot: G2

Faculty Name: ANNAPURNA JONNALAGADDA

1. Title of the project:

"Optimized Meal Planning Using Resource Allocation Algorithms"

2. Team members:

22BBS0019 - Makam Girish

22BBS0237 - SKM Shanush

22BBS0242 - Srisivan K

3. Division of work among team members:

1. Makam Girish: Worked on the Brute Force algorithm along with identifying gaps in existing algorithms.
2. Shanush: Worked on the Greedy algorithm while investigating how this idea can be applied to the real world with a much wider scope and how to enhance it.
3. Srisivan K: Worked on the dynamic programming algorithm. Collaborated with other team members to calculate the time complexities of the algorithms.

4. Problem statement:

Develop a digital personal dietitian application that utilises a Resource Allocation Algorithm to generate optimized, balanced meal plans tailored to individual users. The application will calculate the total caloric intake based on body measurements (height, weight, and age) and automatically adjust macronutrient ratios and dietary preferences. It aims to meet the user's specific dietary requirements while maintaining nutritional balance and supporting personalized diet goals.

5. Introduction:

a. Motivation:

- i. **Rising Demand for Personalized Nutrition:** Growing demand for tailored diet plans based on individual needs.
- ii. **Accessibility:** Professional dietitians can be costly and inaccessible for many.
- iii. **Health Challenges:** Many struggle to maintain balanced diets, leading to health issues.
- iv. **Tech Solution:** Technology can make personalized meal planning more affordable and efficient.
- v. **Data-Driven Health:** Body measurements can help create optimized, balanced meal plans.
- vi. **Convenience:** A digital dietitian provides easy, on-demand access to balanced nutrition.

b. Significance:

- i. Maintaining a balanced diet to meet nutritional needs is important but difficult for many people.
- ii. Hiring a professional dietitian can be expensive, making it inaccessible for many.
- iii. This system provides tailored meal planning based on the customer's body parameters.
- iv. Customization ensures accuracy in achieving macronutrient and calorie targets.
- v. The algorithm improves overall health and helps with adherence to dietary goals.
- vi. Saves time and effort for users in planning and maintaining a balanced diet.

c. Scope and applications:

- i. **Health and Wellness:** Ensures users receive balanced, personalized meal plans for improved health and fitness.
- ii. **Cost-Effective Diet Management:** Provides a low-cost alternative to hiring a personal dietitian.
- iii. **Customized Nutrition:** Generates plans based on individual body metrics (weight, age, activity level) for tailored nutritional recommendations.

- iv. **Dietary Tracking:** Can be integrated with fitness apps to adjust meal portions based on physical activity.
- v. **Time-Efficient Meal Planning:** Reduces the effort of manual meal planning by automatically suggesting portion sizes and balanced meals.

6. Solutions using various strategies

- a. **Describe the strategy**
- b. **Discuss the corresponding algorithm**
- c. **Analyse its complexity.**

1. Brute Force Strategy

Strategy Description

The brute force approach ensures that each food item is allocated at least once, then iteratively adds items or halves the drink portion to approach the calorie target as closely as possible. This approach lacks optimization techniques but tries to exhaustively fill the calorie target by incrementing units until the calories are approximately met.

Algorithm Outline

1. **Initialize:** Deduct one unit's worth of calories for each food item from the total breakfast calorie target.
2. **Allocate Drink:** If enough calories are left, allocate one full or half serving of the drink.
3. **Add Units to Items:** For each food item, attempt to add the maximum possible units without exceeding the remaining calories.

4. **Check Remaining Calories:** After each allocation, check if adding more would exceed the calorie target and stop once it's close.

Algorithm:

1. Initialise `remaining_calories = breakfast_calories`.
2. Allocate at least 1 unit for each food item:
 - a. For each food item in `food_calories`:
 - i. Set `allocation[food] = 1`.
 - ii. Subtract `calorie_per_unit` from `remaining_calories`.
3. Try to allocate the drink:
 - a. If `remaining_calories >= drink_calories_full`, set `allocation["drink"] = 1`.
 - b. Else if `remaining_calories >= drink_calories_half`, set `allocation["drink"] = 0.5`.
 - c. Otherwise, set `allocation["drink"] = 0`.
4. Redistribute remaining calories among food items:
 - a. For each food item in `food_calories`:
 - i. Calculate `max_additional_units = floor(remaining_calories / calorie_per_unit)`.
 - ii. Add `max_additional_units` to `allocation[food]`.
 - iii. Subtract used calories from `remaining_calories`.
5. Return `allocation` and `drink_item`.

Complexity Analysis

1. **Step 2:** Iterate through all n food items once to allocate 1 unit each: $O(n)$.
2. **Step 3:** Check drink allocation, constant time: $O(1)$
3. **Step 4:** Redistribute remaining calories among n food items:
 - For each food, calculate `max_additional_units` ($O(1)$).
 - Total: $O(n)$

Overall Time Complexity: $O(n)$

The brute force approach works for small sets of items but becomes inefficient as the number of items or allowable units grows due to repetitive calculations.

2. Greedy Strategy

Strategy Description

The greedy approach sorts items by calorie density (calories per unit) and prioritises higher-calorie items first. It allocates the maximum possible units of each high-calorie item within the calorie target, leading to an efficient but sometimes imbalanced distribution.

Algorithm Outline

1. **Sort Items by Calorie Density:** Rank all food items and drink based on calories per unit in descending order.
2. **Allocate Units:** Start with the highest-calorie item and allocate the maximum units possible without exceeding the calorie target.
3. **Proceed to the Next Item:** Move to the next item in the sorted list and allocate units until the calorie target is approached as closely as possible.
4. **Stop Allocation:** Once adding any more would exceed the calorie target, halt the allocation.

Algorithm:

- **Input:** food_items[] (array of n food items with calories/unit), drink_calories, calorie_target
- **Output:** allocation (greedy allocation of items)

1. Initialize allocation dictionary with 0 for each item.
2. Compute calorie density for each item: $\text{density}[\text{item}] = \text{calories/unit}$
3. Sort all items (including drink) by calorie density in descending order.
4. $\text{remaining_calories} = \text{calorie_target}$
5. For each item i in sorted list:
 - a. $\text{max_units} = \text{floor}(\text{remaining_calories} / \text{item_calories})$
 - b. $\text{allocation}[\text{item}] = \text{max_units}$
 - c. $\text{remaining_calories} -= \text{max_units} * \text{item_calories}$

6. Return allocation.

Complexity Analysis

1. **Step 1:** $O(n)$ to initialise allocation.
2. **Step 2:** $O(n)$ to compute calorie densities.
3. **Step 3:** $O(n \log n)$ for sorting.
4. **Step 5:** $O(n)$ for iterating through the sorted list.

Thus, the overall **time complexity** = **$O(n \log n)$** .

This approach is efficient due to its linear allocation phase post-sorting but may not always achieve the best distribution because it prioritises calories without considering the calorie balance across items.

Toy Problem

- **Input:**

1. Food items and their calories per unit:
 - Bread: 60 calories per unit
 - Eggs: 120 calories per unit
 - Fruits: 50 calories per unit
 - Cheese: 80 calories per unit
2. Drink and its calories per unit:
 - Juice: 90 calories per unit
3. Calorie target: **500 calories**

- **Steps to Solve:**

1. **Calculate calorie density (calories/unit) for all items:**
 - Bread: $60/1 = 60$

- Eggs: $120/1 = 120$
- Fruits: $50/1 = 50$
- Cheese: $80/1 = 80$
- Juice: $90/1 = 90$

2. Sort items by calorie density in descending order:

- Eggs: 120
- Juice: 90
- Cheese: 80
- Bread: 60
- Fruits: 50

3. Start allocating items while staying within the calorie target:

- Remaining calories: 500

4. Item: Eggs (120 calories/unit):

- Maximum units = $\lfloor 500/120 \rfloor = 4$ $\lfloor 500 / 120 \rfloor = 4$
- Calories used = $4 \times 120 = 480$ $4 \times 120 = 480$
- Remaining calories = $500 - 480 = 20$ $500 - 480 = 20$

5. Item: Juice (90 calories/unit):

- Maximum units = $\lfloor 20/90 \rfloor = 0$ $\lfloor 20 / 90 \rfloor = 0$
- Calories used = $0 \times 90 = 0$ $0 \times 90 = 0$
- Remaining calories = 20

6. Item: Cheese (80 calories/unit):

- Maximum units = $\lfloor 20/80 \rfloor = 0$ $\lfloor 20 / 80 \rfloor = 0$
- Calories used = $0 \times 80 = 0$ $0 \times 80 = 0$
- Remaining calories = 20

7. Item: Bread (60 calories/unit):

- Maximum units = $\lfloor 20/60 \rfloor = 0$ $\lfloor 20 / 60 \rfloor = 0$
- Calories used = $0 \times 60 = 0$ $0 \times 60 = 0$
- Remaining calories = 20

8. Item: Fruits (50 calories/unit):

- Maximum units = $\lfloor 20/50 \rfloor = 0$ $\lfloor 20 / 50 \rfloor = 0$
- Calories used = $0 \times 50 = 0$ $0 \times 50 = 0$

- Remaining calories = 20

9. Final Allocation:

- Eggs: 4 units
- Juice: 0 units
- Cheese: 0 units
- Bread: 0 units
- Fruits: 0 units

● **Output:**

● **Allocation:**

- Eggs: 4 units ($4 \times 120 = 480$ calories)
- Juice: 0 units
- Cheese: 0 units
- Bread: 0 units
- Fruits: 0 units

● **Total Calories Used: 480**

● **Remaining Calories: 20**

3. Dynamic Programming Strategy

Strategy Description

Dynamic programming (DP) is used here as a type of subset-sum approach, where each allocation of units is treated as a possible “state” with the goal of reaching the calorie target. This method systematically explores combinations of items and units to maximise calorie allocation while staying under the target.

Algorithm Outline

- 1. Define Subproblems:** Set up a 2D DP table where each entry $dp[i][j]$ represents the maximum calories achievable with the first i items and a calorie limit of j .
- 2. Initialize Table:** Begin with zero calories allocated.
- 3. Fill DP Table:**

- For each food item, consider adding units up to the maximum calories allowable without exceeding the calorie target.
 - For each calorie limit j , update $dp[i][j]$ as the maximum achievable calories by either including or excluding the current item.
4. **Backtrack Solution:** After filling the table, backtrack from $dp[n][target]$ to determine the optimal unit allocation for each item.

Algorithm:

- **Input:** `food_items[]` (array of n food items with calories/unit), `drink_calories`, `calorie_target`
 - **Output:** allocation (optimal allocation of items)
1. Initialize DP table $dp[n+1][calorie_target+1]$ to 0. $dp[i][j]$ = max calories achievable with first i items and calorie limit j .
 2. For each item i in `food_items`:
 - a. For each calorie limit j (1 to `calorie_target`):
 - i. Exclude the item: $dp[i][j] = dp[i-1][j]$
 - ii. Include the item: $dp[i][j] = \max(dp[i-1][j], dp[i-1][j - k * item_calories] + k * item_calories)$
 - where k is the number of units that fit in the calorie limit j .
 3. Backtrack to determine the optimal allocation:
 - a. Start from $dp[n][calorie_target]$ and trace decisions for each item.
 4. Return allocation.

Complexity Analysis

1. **Step 1:** Initialising the table requires $O(n \cdot C)$, where C is the calorie target.

2. **Step 2:**

- a. For each item ($O(n)$):
 - i. Iterate through calorie limits ($O(C)$):
 1. For each calorie limit, checking all unit counts up to U : $O(U)$.
- b. Total: $O(n \cdot C \cdot U)$.

3. **Step 3:** Backtracking is $O(n)$.

Thus, the overall **time complexity** = $O(n \cdot C \cdot U)$.

This approach is computationally more expensive than the other two, especially for large calorie targets, but it provides the most balanced and close-to-optimal solution.

Toy Problem:

Input:

- **Breakfast Calorie Target:** 400 cal
- **Food Items and Calories per Unit:**
 - Bread: 50 cal/unit
 - Eggs: 100 cal/unit
 - Fruits: 40 cal/unit
 - Cheese: 60 cal/unit
- **Drink:** Juice: 80 cal/unit

Execution Steps:

1. **Step 1:** Define the DP state:
 - a. $dp[i][c]$: Maximum calories achievable using the first i items with a calorie limit of c .
2. **Step 2:** Build the DP table:
 - a. Table size: $(n+1) \times (\text{calories}+1) = 5 \times 401$.

- b. Initialize $dp[0][c] = 0$.
3. **Step 3:** Populate the table for each item and calorie limit:
- a. Bread: $dp[1][c] = \max(dp[0][c], dp[0][c-50] + 50)$
 - b. Eggs: $dp[2][c] = \max(dp[1][c], dp[1][c-100] + 100)$
 - c. Fruits: $dp[3][c] = \max(dp[2][c], dp[2][c-40] + 40)$
 - d. Cheese: $dp[4][c] = \max(dp[3][c], dp[3][c-60] + 60)$
 - e. Juice: $dp[5][c] = \max(dp[4][c], dp[4][c-80] + 80)$
4. **Step 4:** Backtrack from $dp[5][400]$ to find selected items.

Output Allocation (Backtracking):

- Start at $dp[5][400] = 400$
- Juice: 1 unit ($400 - 80 = 320$).
- Cheese: 1 unit ($320 - 60 = 260$).
- Eggs: 1 unit ($260 - 100 = 160$).
- Fruits: 2 units ($160 - 40 \times 2 = 80$).
- Bread: 1 unit ($80 - 50 = 30$).

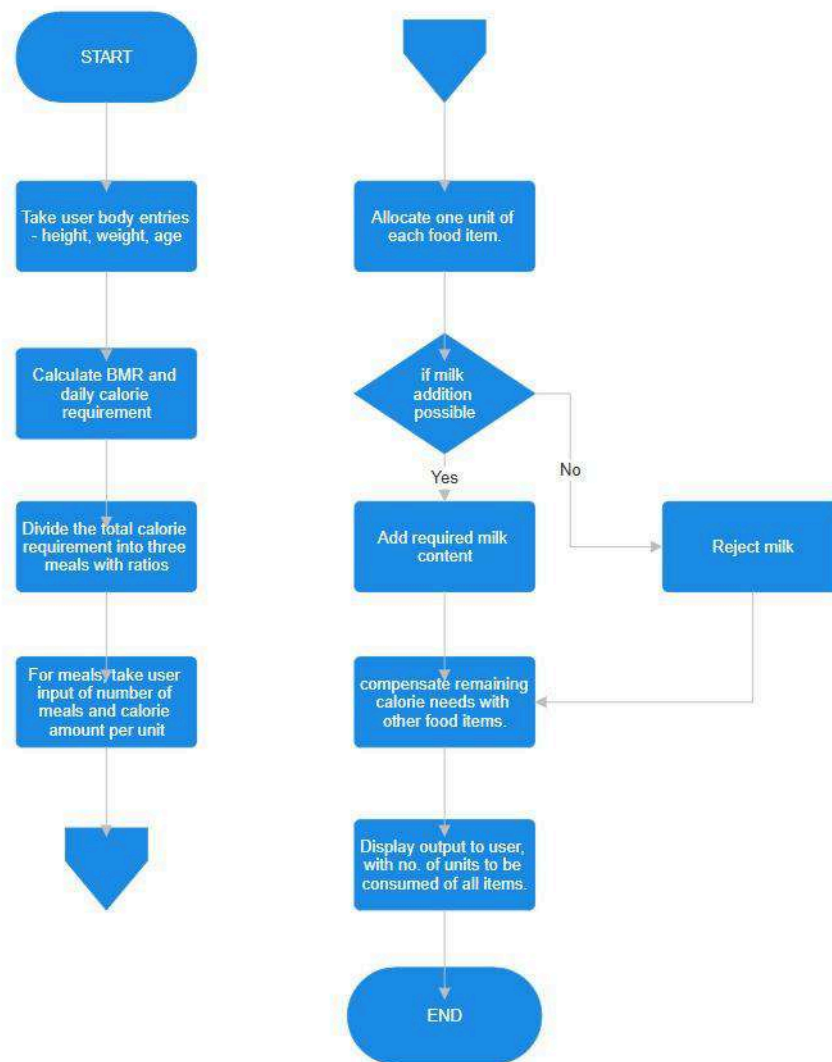
Final Allocation:

- Bread: 1 unit (50 cal).
- Eggs: 1 unit (100 cal).
- Fruits: 2 units ($40 \times 2 = 80$ cal).
- Cheese: 1 unit (60 cal).
- Juice: 1 unit (80 cal).

Total Calories Used: $50 + 100 + 80 + 60 + 80 = 370$ (30 cal remaining).

7. Implementation

a. Framework/ Architecture/ Flow chart



b. Toy example (For Brute-Force Algorithm)

Inputs:

- **Age:** 25 years
- **Weight:** 70 kg
- **Height:** 175 cm

- **Meal items:**
 - **Food items:**
 1. Oatmeal (100 cal per unit)
 2. Eggs (150 cal per unit)
 - **Drink:**
 1. Coffee (50 cal per glass)

Step 1: Calculate Total Daily Calories

We'll start by calculating the total daily calorie requirement using the Harris-Benedict equation.

The formula for BMR (for men) is:

$$\text{BMR} = 88.362 + (13.397 \times \text{weight in kg}) + (4.799 \times \text{height in cm}) - (5.677 \times \text{age})$$

Substitute the values:

$$\text{BMR} = 88.362 + (13.397 \times 70) + (4.799 \times 175) - (5.677 \times 25)$$

$$\text{BMR} = 88.362 + 937.79 + 839.825 - 141.925$$

$$\text{BMR} \approx 1724.052 \text{ calories/day}$$

Now, we account for a sedentary activity level (multiplied by 1.2):

$$\text{Total Daily Calories} = 1724.052 \times 1.2 \approx 2068.862 \text{ calories/day}$$

Step 2: Divide Calories into Meals

The algorithm splits the total daily calories into three meals:

- **Breakfast:** 30% of total calories
- **Lunch:** 40% of total calories
- **Dinner:** 30% of total calories

For breakfast, we allocate:

$$\text{Breakfast Calories} = 2068.862 \times 0.3 \approx 620.66 \text{ calories}$$

Step 3: Meal Plan Allocation for Breakfast

Now, let's distribute the breakfast calories across food and drink items. We will use the breakfast calories (620.66) and the food items to allocate portions:

- **Food items:**
 1. Oatmeal: 100 calories per unit
 2. Eggs: 150 calories per unit
- **Drink:**
 1. Coffee: 50 calories per glass

Step 3a: Allocate at least one unit of each food

- Oatmeal: 1 unit (100 calories)
- Eggs: 1 unit (150 calories)

Remaining Calories:

$$620.66 - 100 - 150 = 370.66 \text{ calories remaining}$$

Step 3b: Try to allocate the drink

- Coffee: 1 glass (50 calories)

Remaining Calories:

$$370.66 - 50 = 320.66 \text{ calories remaining}$$

Step 3c: Redistribute remaining calories among solid foods

- We have 320.66 calories left. Oatmeal has 100 calories per unit, and eggs have 150 calories per unit.
 - Max additional oatmeal units = $\lfloor 320.66/100 \rfloor = 3$ units.
 - Max additional eggs units = $\lfloor 320.66/150 \rfloor = 2$ units.

After adding these additional units:

- Oatmeal: 3 additional units (100 calories each) = 300 calories
- Eggs: 2 additional units (150 calories each) = 300 calories

This exceeds the remaining 320.66 calories, so we stop after adding 1 more oatmeal unit and 1 more egg unit:

- Additional oatmeal: 1 unit (100 calories)
- Additional eggs: 1 unit (150 calories)

Remaining Calories:

$$320.66 - 100 - 150 = 70.66 \text{ calories remaining}$$

Now we have 1 more oatmeal and 1 more egg, so no further food can be added.

Final Allocation for Breakfast

- **Oatmeal:** 4 units ($4 \times 100 = 400$ calories)
- **Eggs:** 2 units ($2 \times 150 = 300$ calories)
- **Coffee:** 1 glass (50 calories)

Total Calories for Breakfast:

$$400 \text{ (Oatmeal)} + 300 \text{ (Eggs)} + 50 \text{ (Coffee)} = 750 \text{ calories}$$

c. Program

Github Repository Link: [Meal Planning Project](#)

8. References

[An Algorithm to Generate a Diet Plan to Meet Specific Nutritional Requirements](#)

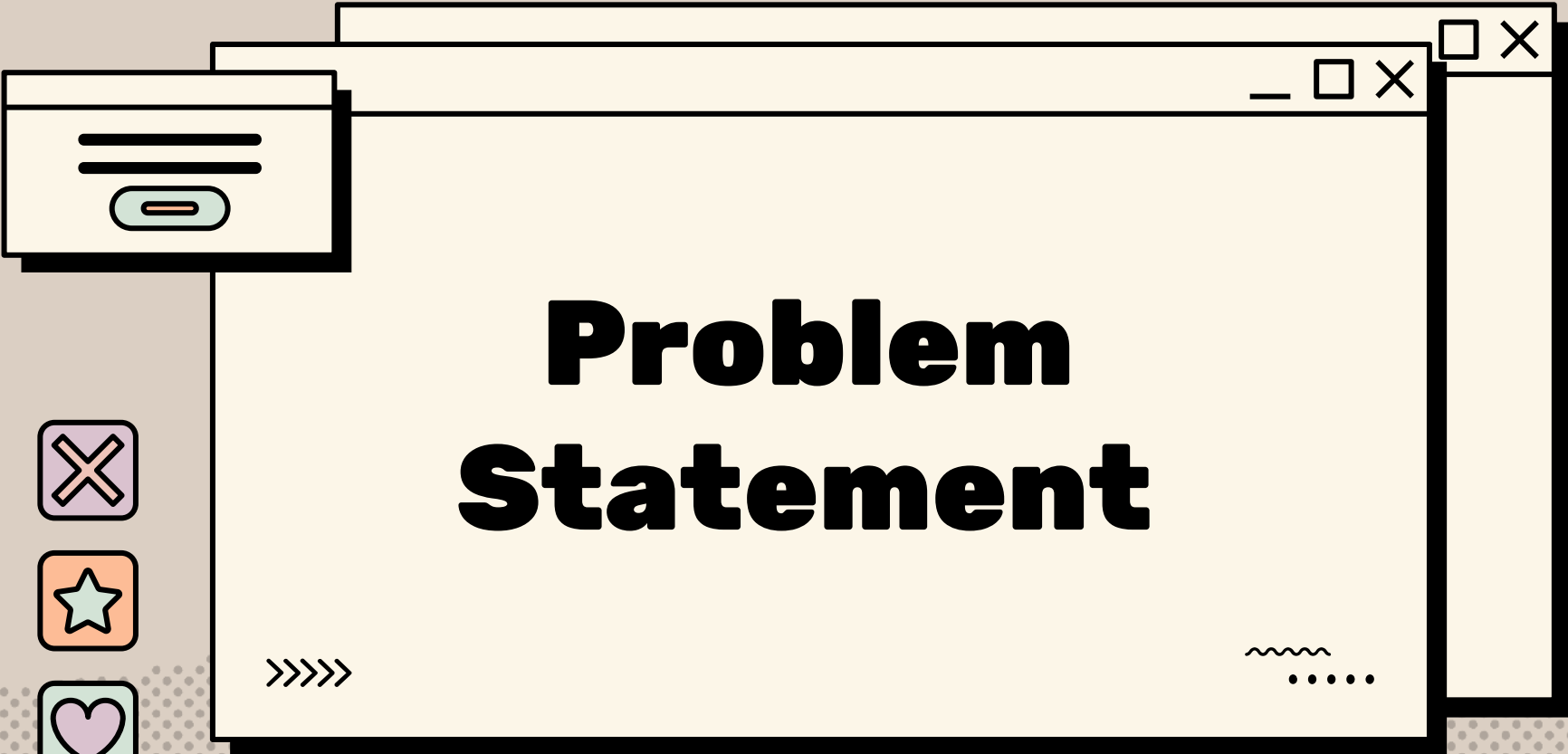


Optimized Meal Planning Using Resource Allocation Algorithm

>>>>>

~~~~~  
.....





# Problem Statement



# Problem Statement



## What is the problem?

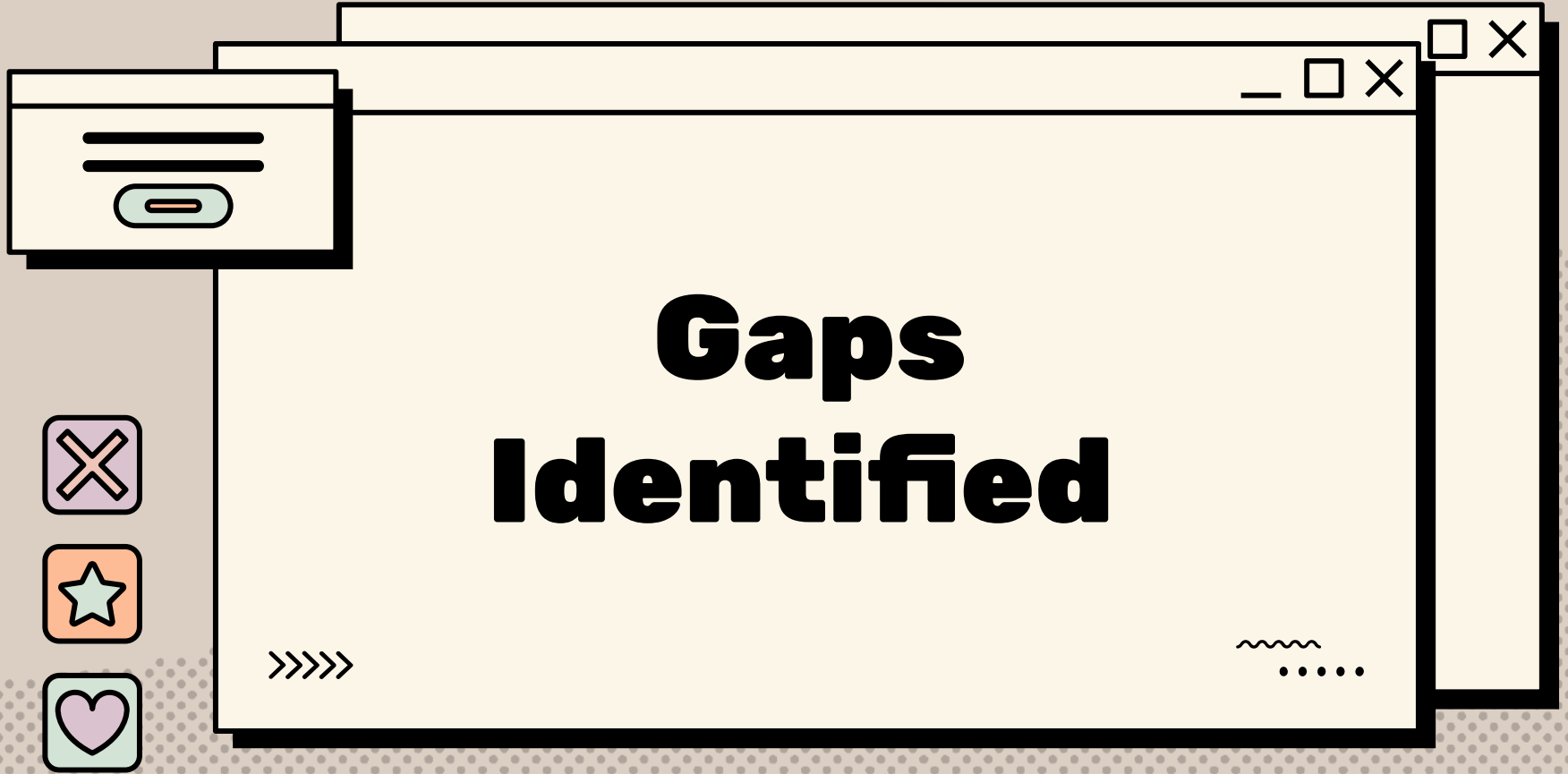
Current meal planners lack customization or are costly.

## Who has the problem?

Individuals wanting personalized, affordable meal plans.

## Why is it important?

To offer accessible, tailored meal planning for balanced nutrition.





# Gaps Identified



## High Cost

Custom meal planning services are often expensive.

## No Personalization

Most tools don't adapt to individual body metrics and needs.

## Incorrect Portions

Some tools provide incorrect portion sizes.

## Limited Variety

Meal plans often lack variety, leading to repetition.



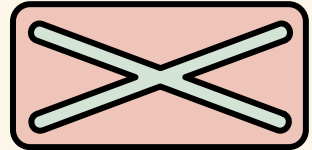
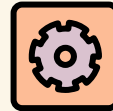
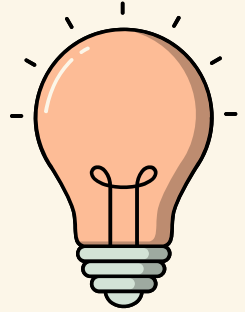
# Strategy and Algorithm Analysis

>>>>

~~~~~  
.....

Strategies Used

- **Brute Force**
 - Tries all possible solutions.
- **Greedy**
 - local optimal choices.
- **Dynamic Programming**
 - Solves subproblems once and reuses solutions



Brute Force - Algorithm

```
function brute_force_allocation(age, weight, height, food_items, drink_item):
    total_calories = calculate_total_calories(age, weight, height)
    breakfast_calories = total_calories * 0.3

    allocation = {}
    remaining_calories = breakfast_calories

    for each food in food_items:
        allocation[food] = 1
        remaining_calories -= food.calories

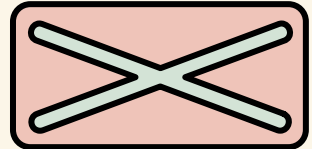
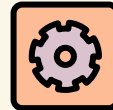
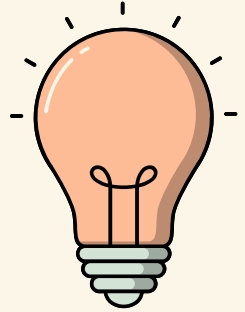
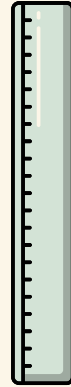
    if remaining_calories >= drink_item.calories:
        allocation["drink"] = 1
    elif remaining_calories >= drink_item.calories / 2:
        allocation["drink"] = 0.5

    for each food in food_items:
        max_units = floor(remaining_calories / food.calories)
        allocation[food] += max_units
        remaining_calories -= max_units * food.calories

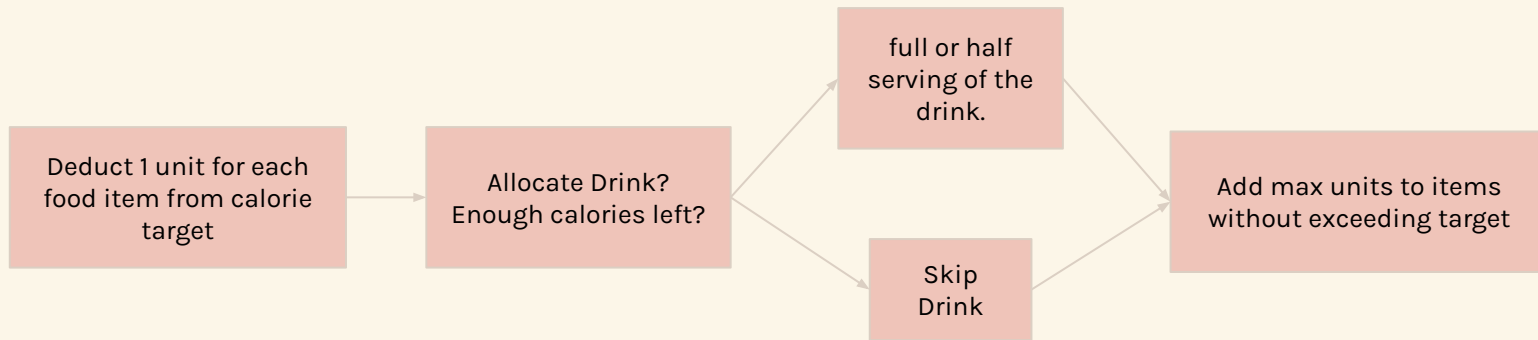
    return allocation
```

Brute Force - Algorithm Outline

- **Initialize:** Deduct one unit's worth of calories for each food item from the total breakfast calorie target.
- **Allocate Drink:** If enough calories are left, allocate one full or half serving of the drink.
- **Add Units to Items:** For each food item, attempt to add the maximum possible units without exceeding the remaining calories.
- **Check Remaining Calories:** After each allocation, check if adding more would exceed the calorie target and stop once it's close.



Brute Force - Flow Diagram





Toy Example - Input

Age: 25 years

Weight: 70 kg

Height: 175 cm



Toy Example - Food Items

Egg: 78 calories

Toast: 120 calories

Bacon: 92 calories

Drink - Orange juice
calories per glass

Toy Example - Calculations

The Harris-Benedict Equation



Men:

$$\text{BMR} = 88.362 + (13.397 \times \text{weight in kg}) + (4.799 \times \text{height in cm}) - (5.677 \times \text{age in years})$$



Women:

$$\text{BMR} = 447.593 + (9.247 \times \text{weight in kg}) + (3.098 \times \text{height in cm}) - (4.330 \times \text{age in years})$$

Toy Example - Calculations

- **BMR Calculation:** Using the Harris-Benedict equation for men:
 - $\text{BMR} = 88.362 + (13.397 \times 70) + (4.799 \times 175) - (5.677 \times 25) = 1724.25$ calories
- **Daily Calories:**
 - $\text{Total Calories} = 1724.25 \times 1.2 = 2069.1$ calories



Toy Example - Calculations

- **Meal Division**
 - **Breakfast Calories:** $2069.1 \times 0.3 = 620.73$ calories
 - **Lunch Calories:** $2069.1 \times 0.4 = 827.64$ calories
 - **Dinner Calories:** $2069.1 \times 0.3 = 620.73$ calories

Toy Example - Output

```
--- Optimized Breakfast Meal Plan (Brute Force) ---  
Total breakfast calories target: 621.00 cal  
Egg: 5 unit(s) (390 cal)  
Toast: 1 unit(s) (120 cal)  
Bacon: 1 unit(s) (92 cal)  
Orange juice: 0.5 glass(es) (55 cal)
```

Greedy - Algorithm

```
function greedy_allocation(age, weight, height, food_items, drink_item):
    total_calories = calculate_total_calories(age, weight, height)
    breakfast_calories = total_calories * 0.3

    food_items.sort_by_calories(descending)

    allocation = {}
    remaining_calories = breakfast_calories

    for each food in food_items:
        max_units = floor(remaining_calories / food.calories)
        allocation[food] = max_units
        remaining_calories -= max_units * food.calories

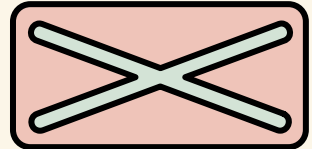
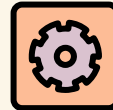
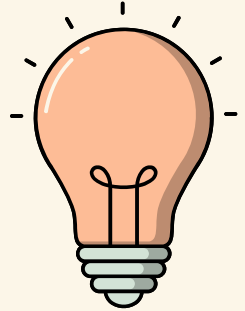
    if remaining_calories >= drink_item.calories:
        allocation["drink"] = 1
    elif remaining_calories >= drink_item.calories / 2:
        allocation["drink"] = 0.5

    return allocation
```

Greedy - Algorithm Outline

- **Sort Items:** Rank food and drink by calorie density (highest first).
- **Allocate Units:** Start with the highest-calorie item and add units without exceeding the target.
- **Proceed:** Move to the next item and continue until the target is nearly reached.

Stop: Halt when adding more would exceed the target.



Greedy - Flow Diagram



Greedy - Algorithm

```
function dp_allocation(age, weight, height, food_items, drink_item):
    total_calories = calculate_total_calories(age, weight, height)
    breakfast_calories = total_calories * 0.3
    max_calories = floor(breakfast_calories)

    dp = array of (items + 1) x (max_calories + 1)

    for each item:
        for each calorie capacity:
            dp[i][j] = max(dp[i-1][j], dp[i-1][j - item.calories] + item.calories)

    remaining_calories = max_calories
    allocation = {}

    for each item in dp:
        if dp[i][remaining_calories] != dp[i-1][remaining_calories]:
            allocation[item] = 1
            remaining_calories -= item.calories

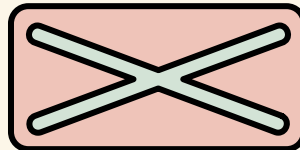
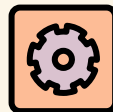
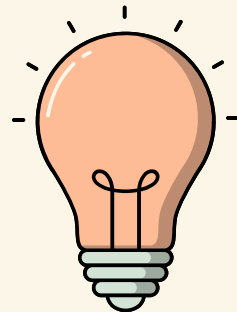
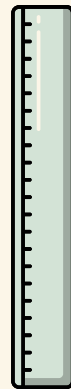
    if remaining_calories >= drink_item.calories:
        allocation["drink"] = 1
    elif remaining_calories >= drink_item.calories / 2:
        allocation["drink"] = 0.5

    return allocation
```

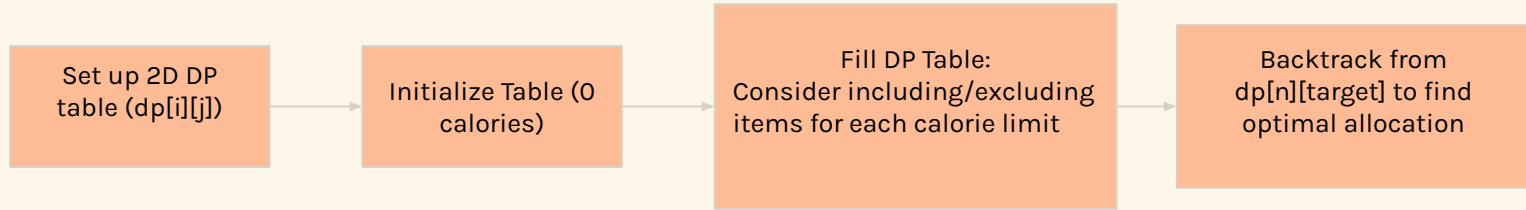
Dynamic Programming

Algorithm outline:

- **Define Subproblems:** Create a 2D DP table where each entry $dp[i][j]$ represents the max calories with the first i items and calorie limit j .
- **Initialize Table:** Start with zero calories.
- **Fill DP Table:** For each item, update $dp[i][j]$ by considering whether to include or exclude the item within the calorie limit.
- **Backtrack Solution:** After filling the table, backtrack from $dp[n][target]$ to find the optimal allocation.



Dynamic Programming - Flow Diagram



Dynamic Programming - Algorithm

```
function dp_allocation(age, weight, height, food_items, drink_item):
    total_calories = calculate_total_calories(age, weight, height)
    breakfast_calories = total_calories * 0.3
    max_calories = floor(breakfast_calories)

    dp = array of (items + 1) x (max_calories + 1)

    for each item:
        for each calorie capacity:
            dp[i][j] = max(dp[i-1][j], dp[i-1][j - item.calories] + item.calories)

    remaining_calories = max_calories
    allocation = {}

    for each item in dp:
        if dp[i][remaining_calories] != dp[i-1][remaining_calories]:
            allocation[item] = 1
            remaining_calories -= item.calories

    if remaining_calories >= drink_item.calories:
        allocation["drink"] = 1
    elif remaining_calories >= drink_item.calories / 2:
        allocation["drink"] = 0.5

    return allocation
```


Complexity Analysis

Strategy	Time Complexity	Space Complexity	Pros	Cons
Brute Force	$O(U^n * n)$	$O(n)$	Simple and easy to implement	Inefficient for larger inputs
Greedy	$O(n \log n)$	$O(n)$	Fast due to prioritization	May miss balanced allocation
Dynamic Program	$O(n * C * U)$	$O(n * C)$	Finds the best allocation	Higher complexity and memory usage

1. Let **n** be the number of food items and **m** the max units allocable per item within the calorie target.
2. **C** is the calorie target.
3. **U** = max_units = $\lfloor \text{remaining calories} / \text{calories_per_unit} \rfloor$



Thanks!

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik**

Please keep this slide for attribution