Clicker Question #1

How can we best describe the runtime of BruteForceChange?

A. Linear
B. Quadratic
C. Polynomial (but not linear or quadratic)
D. Exponential
E. Factorial

Clicker Question #1

How can we best describe the runtime of BruteForceChange?

A. Linear
B. Quadratic
C. Polynomial (but not linear or quadratic)
D. Exponential
E. Factorial

Dynamic Programming

CPSC 320 2024S

Optimization problems

- Just as with Greedy algorithms, this unit primarily deals with **optimization problems**:
  - We have a problem with several valid solutions
  - There is a score, or objective function, that tells us how good (or bad) each solution is
  - We are looking for the valid solution that minimizes or maximizes the score

When to use dynamic programming

- A ~~greedy~~ dynamic programming algorithm proceeds by:
  - ~~Making a single choice based on a simple, local criterion~~
  - Considering a (polynomial) number of possible choices
  - Solving one or more subproblem(s) that result from each choice
  - Combining the choice and its associated subproblem(s) (e.g., by calculating the total score)
  - Selecting the best choice (with maximum or minimum score)
- This technique is useful when the possible choices lead to **repeated or overlapping** subproblems

When to use dynamic programming?

- **Repeated/overlapping subproblems:**
  - Meaning that if two of the subproblems are **A** and **B** then smaller subproblems of **A** are also smaller subproblems of **B**
- Choosing an algorithm design technique:
  - If you have a simple, local criterion to make a choice that leads to an optimal (or good enough!) solution → **Greedy**
  - If not, and subproblems overlap → **Dynamic programming**
  - If subproblems don't overlap → **Divide and conquer**

Dynamic programming versus memoization?

- These are variants of the same problem-solving approach
- In this class we use memoization to refer to to a *recursive* implementation (top-down approach)
- We use dynamic programming to refer to an *iterative* implementation (bottom-up approach)

An interesting question is, "Where did the name, dynamic programming, come from?" The 1950s were not good years for mathematical research. […] Wilson […] was Secretary of Defense […] I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics […] I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought […] it's impossible to use the word dynamic in a pejorative sense. […] Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to.

—Richard Bellman, Eye of the Hurricane: An Autobiography (1984, page 159)

See https://en.wikipedia.org/wiki/Dynamic_programming

Designing a DP or Memoization algorithm

- Step 1: determine the possible choices (and associated subproblems)
  - E.g., which intervals from **1** to **j** to choose in the weighted interval scheduling problem
  - E.g., which coins to give in the change-making problem (current worksheet!)
  - E.g., which items to choose for the knapsack problem

Designing a DP or Memoization algorithm

- Step 2: define a recurrence relation for **the value of the objective function** in terms of **its value(s) for one of more subproblems**
- E.g., OPT(j) = min{OPT(i-25), OPT(i-10), OPT(i-1)}
- E.g., HP[i,j] = max{1, min{HP[i+1,j], HP[i,j+1]- G[i,j]} (in upcoming tutorial and assignment)

  (base cases omitted here)
- A **table** is used to store values of the objective function for different subproblems

Designing a DP or Memoization algorithm

- Step 3: determine the “shape” of the table that stores solutions to subproblems
  - Dimension will depend on the number and range of parameters in the subproblem
  - It’s usually an array with as many dimensions as there are parameters to the subproblem
    - E.g., M[1, …, n] for weighted interval scheduling
    - E.g., M[0, …, n][0, …, W] for knapsack

Designing a DP or Memoization algorithm

- Step 4: implement the algorithm
  - Approach 1: memoization (recursion)
    - Like a straightforward recursive approach to compute the value given by the recurrence relation
    - But first look at the table to see if this solution has already been computed
    - If so, return it immediately

Designing a DP or Memoization algorithm

- Approach 2: iteration (sometimes only called “dynamic programming”)
  - Loop over the problem, starting with the base cases
  - Then solve later subproblems
  - Make sure that by the time subproblem S needs the solution to S’, the solution to S’ has already been computed
- For each subproblem:
  - Compute the optimal solution to the subproblem, using table lookups instead of recursive calls

Designing a DP or Memoization algorithm

- Step 5: retrieve the optimal solution
  - Step 4 gives us the value of the objective function for the optimal solution, but not the solution
  - To retrieve the solution:
    - Start with the original problem and determine the last choice we made
    - Insert this choice into the solution (usually at the end)
    - Then repeat starting from the subproblem we get after making that choice
- We don’t want to store solutions for every subproblem because it would increase the space

A non-optimization example: Fibonacci numbers

- Each Fibonacci number is the sum of the two previous Fibonacci numbers
  - The sequence: 1, 1, 2, 3, 5, 8, 13, 21, …
- Recurrence:
  - FIB(0) = 0, FIB(1) = 1 // base cases
  - FIB(n) = FIB(n-1) + FIB(n-2) // recursive cases

A straightforward recursive implementation

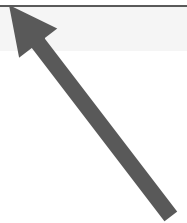
```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

A memoized implementation

```
# memoized version
def fibonacci_mem(n, memo={}):
    if n in memo:
        return memo[n]
    elif n <= 1:
        return n
    else:
        memo[n] = fibonacci_mem(n-1, memo) + fibonacci_mem(n-2, memo)
        return memo[n]
```

An iterative implementation – clicker question

```
# iterative dynamic programming version
def fibonacci_iter(n):
    fib = [0, 1]  # Initialize Fibonacci sequence with the first two numbers
    for i in range(2, n + 1):
        fib.append(                    )
    return fib[n]
```



Fill in the blank!

An iterative implementation

```python
# iterative dynamic programming version
def fibonacci_iter(n):
    fib = [0, 1]  # Initialize Fibonacci sequence with the first two numbers
    for i in range(2, n + 1):
        fib.append(fib[i - 1] + fib[i - 2])
    return fib[n]
```