Algorithm Design Techniques

Greedy Algorithms
Dynamic Programming

Greedy Algorithms

- Choose the best option (that we can see at the time) during each phase
  - Dijkstra
  - Prim
  - Kruskal

- These three all gave an optimal solution; nice, but not all greedy algorithms do so

Greedy Algorithms

- Making change for a currency (e.g., penny, nickel, dime, quarter)
  - Goal: fewest coins used

Greedy Algorithms

- Making change for a currency (e.g., penny, nickel, dime, quarter)
  - Goal: fewest coins used
  - Choose largest coin at each round
  - Which coins for 18 cents?

Greedy Algorithms

- Making change for a currency (e.g., penny, nickel, dime, quarter)
  - Goal: fewest coins used
  - Choose largest coin at each round
  - Which coins for 18 cents?
      - 1 dime
      - 1 nickel
      - 3 pennies
  - Does this always work?

Greedy Algorithms

- Making change for a currency (e.g., penny, nickel, dime, quarter)
  - Goal: fewest coins used
  - Choose largest coin at each round
  - Which coins for 18 cents?
    - 1 dime
    - 1 nickel
    - 3 pennies
  - Does this always work?
    - Yes! (for this currency system)

Greedy Algorithms

- Making change for a currency (e.g., penny, nickel, dime, quarter)
  - Goal: fewest coins used
  - Let’s try a different currency (1, 5, 9, and 15 cent coins)
  - Choose largest coin at each round
    - Which coins for 18 cents

Greedy Algorithms

- Making change for a currency (e.g., penny, nickel, dime, quarter)
  - Goal: fewest coins used
  - Let’s try a different currency (1, 5, 9, and 15 cent coins)
  - Choose largest coin at each round
    - Which coins for 18 cents
      - 1 “15” coin
      - 3 “1” coins
  - Does this always work?

Greedy Algorithms

- Making change for a currency (e.g., penny, nickel, dime, quarter)
  - Goal: fewest coins used
  - Let’s try a different currency (1, 5, 9, and 15 cent coins)
  - Choose largest coin at each round
    - Which coins for 18 cents
      - 1 “15” coin
      - 3 “1” coins
  - Does this always work?
    - Optimal is 2 “9” coins
    - Obviously not
- How to solve in the general sense? exhaustive?

Greedy Algorithms – Making Change

- Let's try an exhaustive search...
  - Write a method where we ask it to consider all coins 'c' or smaller, for some 'amount'

```
int[] denoms1 = { 1, 5, 10, 25 };
int[] denoms2 = { 1, 5, 9, 15 };
```

```
public static int requestCoins(int[] denoms, int c, int amount) {
    if (amount == 0) return 0;
    int value = denoms[c];  // Let's get the current denomination
    if (amount < value) {
        return requestCoins(denoms, c - 1, amount);
    }
    // Let's see how many it takes using this denomination
    int current = 1 + requestCoins(denoms, c, amount - value);
    // Let's see how many it takes using only lower denominations
    int lower = requestCoins(denoms, c - 1, amount);

    // Whichever number of coins is smaller, return that
    return Math.min(current, lower);
}
```

Greedy Algorithms – Making Change

- Let’s try an exhaustive search...
  - Write a method where we ask it to consider all coins ‘c’ or smaller, for some ‘amount’

```
int[] denoms1 = { 1, 5, 10, 25 };
int[] denoms2 = { 1, 5, 9, 15 };
```

```
public static int requestCoins(int[] denoms, int c, int amount) {
    if (amount == 0) return 0;
    int value = denoms[c];  // Let's get the current denomination
    if (amount < value) {
        return requestCoins(denoms, c - 1, amount);
    }
    // Let’s see how many it takes using this denomination
    int current = 1 + requestCoins(denoms, c, amount - value);
    // Let’s see how many it takes using only lower denominations
    int lower = requestCoins(denoms, c - 1, amount);

    // Whichever number of coins is smaller, return that
    return Math.min(current, lower);
}
```

  - Problem: We recompute many sub-problems over and over again (remember recursive Fibonacci?)

Greedy Algorithms – Making Change

- ***Memoizing*** – Solving the repeated sub-problem compute problem (a bit of a mouthful to say)
  - An optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.
  - Write a ***memo*** to ourselves when we compute something new, then use that result again when given the same inputs.

- This is a tradeoff of using more memory to gain improved speed

Greedy Algorithms – Making Change – Memoization

```
public static int requestCoins(List<Map<Integer, Integer>> memo, int[] denoms, int c, int amount) {
    if (memo.get(c).containsKey(amount)) {
        return memo.get(c).get(amount);
    }
    if (amount == 0) return 0;
    int value = denoms[c];  // Let's get the current denomination
    if (amount < value) {
        return requestCoins2(memo, denoms, c - 1, amount);
    }
    int current = 1 + requestCoins2(memo, denoms, c, amount - value);
    int lower = requestCoins2(memo, denoms, c - 1, amount);

    memo.get(c).put(amount, Math.min(current, lower));

    return memo.get(c).get(amount);
}
```

Greedy Algorithms – Making Change – Memoization

```
public static void main(String[] args) {
	int[] denoms1 = { 1, 5, 10, 25 };
	int[] denoms2 = { 1, 5, 9, 15 };
	List<Map<Integer, Integer>> memo1 = new ArrayList<Map<Integer, Integer>>();
	List<Map<Integer, Integer>> memo2 = new ArrayList<Map<Integer, Integer>>();
	memo1.add(new HashMap<Integer, Integer>());  // 1
	memo1.add(new HashMap<Integer, Integer>());  // 5
	memo1.add(new HashMap<Integer, Integer>());  // 10
	memo1.add(new HashMap<Integer, Integer>());  // 25

	memo2.add(new HashMap<Integer, Integer>());  // 1
	memo2.add(new HashMap<Integer, Integer>());  // 5
	memo2.add(new HashMap<Integer, Integer>());  // 9
	memo2.add(new HashMap<Integer, Integer>());  // 15

	System.out.println("-- Using Exhaustive --");
	int coins1 = requestCoins(denoms1, 3, 18);
	int coins2 = requestCoins(denoms2, 3, 18);
	System.out.println(coins1);
	System.out.println(coins2);

	System.out.println("-- Using Memo --");
	coins1 = requestCoins(memo1, denoms1, 3, 18);
	coins2 = requestCoins(memo2, denoms2, 3, 18);
	System.out.println(coins1);
	System.out.println(coins2);
}
```

Dynamic Programming

- Solve all smaller problems; to avoid any recomputation
- For each coin (or smaller) and for each amount (up to some max), compute number of coins required
- Then following a recursive approach, use the table of pre-computed values to find the solution

Dynamic Programming

- Solve all smaller problems; to avoid any recomputation
- For each coin (or smaller) and for each amount (up to some max), compute number of coins required
- Then following a recursive approach, use the table of pre-computed values to find the solution

- Doesn’t this sound like computing all possible values and “memoizing” them in advance?
  - Hope so, because that is basically what it is!

Dynamic Programming – Making Change

- Build a table of results
- In our example
  - Rows are the coin denominations, columns are the possible amounts
  - The values are how many coins of that denomination, or smaller, are needed

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5	6
9	1	2	3	4	1	2	3	4	1	2	3	4	5	2	3	4	5	2
15	1	2	3	4	1	2	3	4	1	2	3	4	5	2	1	2	3	2

Dynamic Programming – Making Change

- Make change for 18 cents

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5	6
9	1	2	3	4	1	2	3	4	1	2	3	4	5	2	3	4	5	2
15	1	2	3	4	1	2	3	4	1	2	3	4	5	2	1	2	3	2

Dynamic Programming – Making Change

- Make change for 18 cents
  - Choose largest coin denomination, 15, choose the amount, 18, find the number of coins

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5	6
9	1	2	3	4	1	2	3	4	1	2	3	4	5	2	3	4	5	2
15	1	2	3	4	1	2	3	4	1	2	3	4	5	2	1	2	3	2

Dynamic Programming – Making Change

- Make change for 18 cents
  - Choose largest coin denomination, 15, choose the amount, 18, find the number of coins
  - We choose to use a 15 coin, that leaves us with 3 cents leftover, so we look it up and find we need to use 3 coins, and they all must be 1 cent denominations

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5	6
9	1	2	3	4	1	2	3	4	1	2	3	4	5	2	3	4	5	2
15	1	2	3	4	1	2	3	4	1	2	3	4	5	2	1	2	3	2

Dynamic Programming – Making Change

- Make change for 18 cents
  - If we choose not to use the 15, then move to the 9; its the same as before
  - That tells use the 15 isn't necessary

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5	6
9	1	2	3	4	1	2	3	4	1	2	3	4	5	2	3	4	5	2
15	1	2	3	4	1	2	3	4	1	2	3	4	5	2	1	2	3	2

Dynamic Programming – Making Change

- Make change for 18 cents
  - We choose to use one 9 coin, that leaves us with an amount of 9 left to make change
  - Look up the 9 denomination and the remaining 9 amount, one more coin

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5	6
9	1	2	3	4	1	2	3	4	1	2	3	4	5	2	3	4	5	2
15	1	2	3	4	1	2	3	4	1	2	3	4	5	2	1	2	3	2

Dynamic Programming – Making Change

- Make change for 18 cents
  - If we choose not to use the 15 or 9, then move to the 5; different result
  - That tells use the 9 is necessary (to get the fewest coins used)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5	6
9	1	2	3	4	1	2	3	4	1	2	3	4	5	2	3	4	5	2
15	1	2	3	4	1	2	3	4	1	2	3	4	5	2	1	2	3	2

Dynamic Programming – Making Change

- Make change for 18 cents
  - If we choose to use only one 9, we then lookup denomination 5 and amount 9 for remaining coins
  - We'll need one 5 coin and then four 1 coins

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5	6
9	1	2	3	4	1	2	3	4	1	2	3	4	5	2	3	4	5	2
15	1	2	3	4	1	2	3	4	1	2	3	4	5	2	1	2	3	2

Dynamic Programming – Making Change

- We can see the transition in the table when larger denominations are used

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5	6
9	1	2	3	4	1	2	3	4	1	2	3	4	5	2	3	4	5	2
15	1	2	3	4	1	2	3	4	1	2	3	4	5	2	1	2	3	2

Dynamic Programming – Making Change

- We can see the transition in the table when larger denominations are used

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5	6
9	1	2	3	4	1	2	3	4	1	2	3	4	5	2	3	4	5	2
15	1	2	3	4	1	2	3	4	1	2	3	4	5	2	1	2	3	2