CSC373 A1

February 2022

4.5-2

The running time for Strassen's algorithm is $\Theta(n^{lg7})$. The largest integer value a that satisfies $n^{lg7} > n^{lg(a)}$ is 48.
If we choose $a = 48$, then $T(n) = 48(n/4) + \Theta(n^2)$. From the master theorem, since $log_4(48) > 2$, therefore $n^2 < n^{log_4(48)-\epsilon}$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{log_4(48)})$, which is asymptotically faster than Strassen's algorithm.
Therefore, we have proved that 48 is the largest integer value that Professor Caesar's algorithm would be asymptotically faster than Strassen's algorithm.

4-5

a. Define the total number of bad chips as b, and the number of good chips as $n - b$. Since at least half of the chips are bad chips, therefore we can find a set of bad chips with size $n - b$. Name this set of bad chips as B, and the set of all good chips as G. The chips in B will tell chips in B are good, and chips in G are bad. Although chips in G will tell the correct answers, the answers are opposite to the answers given by chips in B. It is not possible to distinguish. The rest chips other than B and G are also bad chips which tell the incorrect answers. Therefore, the professor cannot necessarily determine which chips are good using this kind of pairwise test.

b. Solution: Step1. Arbitrarily pairwise the chips. (If the total number of the chips is odd, leave alone the last chip.)
Step2. Conduct the pairwise test. If the test result says both chips are good, randomly remove one in the pair; Else, remove both in the pair.
Step3. Recursively conducted the pairwise test among the remaining chips.
Step4.
Condition1: If there is 1 chip left in the last step of recursion. Then this chip is a good chip;
Condition 2: There are 2 chips left, conduct pairwise test between these two chips.
Condition 2.1: if the test result says both are good, then both are good chips
Condition 2.2: else we only know that one of the pair is good, but we can be sure that the last chip left alone in step 1. is a good chip.

This solution reduces the problem to at most half the size recursively, as in step 2.

Proof:
Observation: The pair with result not saying both are good has at least one bad chip. Therefore, in step 2, remove both in the pair will not affect the fact that good chips are at least half of the remaining chips.
Case 1. We have exactly one more good chips than bad chips.
Case 1.1 If the left alone chip in step 1 is a good chip. Then the remaining chips are half good and half bad. From the Observation, at each recursive including the last step, the chips will always be half good and half bad. This corresponds to Condition 2.2 in the solution.
Case 1.2 If the left alone chip is a bad chip. From the Observation, at each recursive including the last step, there will always be more good chips than bad chips. In the last step, the good chip will remain. This corresponds to Condition 1 in the Solution.
Case 2. We have more than one good chips than bad chips. No matter the left alone chip is good or bad, from the Observation, at each recursive including the last step, there will always be more good chips than bad chips. In the last step, the good chip(s) will remain. This corresponds to Condition 1 or Condition 2.1 in the Solution.

c. From the solution in b., the running time to find a good chip is $T(n) \leq T(n/2) + n/2$. By master theorem, $T(n) = O(n)$.
After finding a good chip, we can use this good chip to test the rest chips. This cost $n - 1$ time. Therefore, the total time is $\Theta(n)$

16.2-7

Solution: Sort the two sets A and B in either ascending or descending order.
Running time: Each sorting cost $\Theta(nlogn)$, $T(n) = 2\Theta(nlogn) = \Theta(nlogn)$.

Proof
Define "inversion" in the two sets A and B: (i, j) such that $a_i < a_j$ but $b_i > b_j$.
Observation: There is no "inversion" in A and B if both sets are sorted in ascending or descending order.

Proof maximum payoff by contradiction
Suppose for contradiction that the Solution above is not optimal (does not give the maximum payoff).
Consider an optimal solution S^* with the fewest inversions.
Suppose there is some $i < j$ so that $a_i < a_j$ and $b_i > b_j$. Only consider the contribution to the payoff by a_i, a_j, b_i, b_j, that is $a_i^{b_i} a_j^{b_j}$.
If we swap b_i and b_j, the contribution will be changed to $a_i^{b_j} a_j^{b_i}$.
Since $\frac{a_i^{b_j} a_j^{b_i}}{a_i^{b_i} a_j^{b_j}} = (\frac{a_j}{a_i})^{b_i - b_j}$, which is greater than 1. Therefore, the changed contribution is greater than the previous contribution. This is a contradiction.
Therefore, we proved that the Solution above is optimal (gives the maximum payoff).

16-1

a. Algorithm:
If $n = 0$, choose no coins;
Else, let c be the largest coin with value less or equal to n. Choose this coin.
Recursively solve the sub-problem of making change for $n - c$ cents.

Another similar solution is to choose $\lfloor n_1/c^i \rfloor$ number of c^i, at recursive step i. (c^1 is quarter, c^2 is dime, c^3 is nickle, c^4 is pennie. n_1 is the total amount of cents need be changed at each recursive step.)

Runtime: let k be the number of coins used in the change. There are k recursive steps in total. In each step, the running time at each step is $O(1)$. Therefore the total runtime is $\Theta(k) \in O(n)$.

Proof:
1. Prove that there is an optimal solution for the problem of n cents that contains c (c is the largest coin with value less or equal to n).
Consider an optimal solution.
If this optimal solution contains coin c, then we proved that there is an optimal solution for the problem of n cents that contains c.
If this optimal solution does not contain coin c,
Case 1: $0 < n < 5$, then it must only contain pennies, therefore contains at least one c.
Case 2: $5 \leq n < 10$, if it does not contain c (nickle), it will only contain at least 5 pennies, then we can replace the 5 pennies by a nickle, which decrease the number of coins.
Case 3: $10 \leq n < 25$, similarly, if it does not contain c (dim), the solution must be only composed by nickles and pennies. We can at least replace the nickles and pennies with sum 10 by a dim, which decrease the number of coins.
Case 4: $25 \leq n$, if it does not contain a quarter. Then in one case, it contains three dims, which we can replace by a quarter and a nickle. The other case, it contains at most two dimes, and a subset of dimes, nickles or pennies sum up to 25, which we can also replace by a quarter. The replacement in both cases will decrease the number of coins.
Therefore, we proved that there is an optimal solution for the problem of n cents that contains c.

2. Assume the optimal solution for making change of n uses k coins. We will prove that the optimal solution for making change of $n - c$ cents uses $k - 1$ coins.
Suppose for contradiction that the optimal solution for making change of $n - c$ cents uses less than

$k-1$ coins.
Then by adding the coin c, we can solve the problem of n by less than k coins.
This is a contradiction. Therefore, we proved that the optimal solution for $n-c$ cents will use $k-1$ coins, if the optimal solution for n uses k coins.

Therefore, combine proof 1. and 2., we prove that by recursively select the largest coin with value less or equal to the current total amount, we will obtain the optimal solution making change for n cents.

b. Algorithm:
If $n=0$, choose no coins;
Else, let $c^i, 0 \le i \le k$ be the largest coin with value less or equal to n. Choose this coin.
Recursively solve the sub-problem of making change for $n-c^i$ cents.

Another similar solution is to choose $\lfloor n_1/c^i \rfloor$ number of c^i, at recursive step i. ($0 \le i \le k$, and n_1 is the total amount of cents need be changed at each recursive step.)

Proof
Lemma: Let p_i be the number of $c^i, 0 \le i \le k$ used in the optimal solution, then $p_i < c$ for $0 \le i \le k-1$.
Prove Lemma: Suppose for contradiction that there is a $p_i \ge c$ for $0 \le i \le k-1$. Then $p_i * c^i \ge c * c^i$.
We can use a c^{i+1} to replace the set of c^i coins with size c, which decreases the number of coins used.
This is a contradiction. Therefore, we proved the Lemma.

1. Consider c^m is the largest coin with value less or equal to n, $0 \le m \le k$. Then we will prove that the optimal solution must contain c^m.
Suppose for contradiction that there is an optimal solution S^*, which does not contain c^m.
Then S^* should only contain $c^i, 0 \le i \le m-1$.
The total amount of the coins in solution S^* is

$$\begin{aligned}\sum_{i=0}^{m-1} a_i c^i &< \sum_{i=0}^{m-1} c c^i \text{ (from Lemma)} \\ &= c\frac{c^m-1}{c-1} \\ &< c^m - 1 \\ &< c^m\end{aligned}$$

However, the total amount of the coins in solution should be $\ge c^m$, since $n \ge c^m$. This is a contradiction.
Therefore, we proved that the optimal solution must contain c^m. (c^m is the largest coin with value less or equal to n, $0 \le m \le k$)

2. Similar to a., we can prove that the optimal solution for $n-c^i$ cents will use $a-1$ coins, if the optimal solution for n uses a coins.

Therefore, combine proof 1. and 2., we prove that by recursively select the largest coin with value less or equal to the current total amount, we will obtain the optimal solution making change for n cents.

c. For example, the set of coin denominations quarters, dimes and pennies.
Consider making change for 30 cents. From greedy algorithm, we will first choose a quarter and then five pennies, which uses 6 coins.
However, the optimal solution is to use 3 dimes.
In this case, the greedy algorithm does not yield the optimal solution.

d. We solve this problem by Dynamic Programming.
Suppose $number[i]$ be the fewest number of coins we can use to make change for i cents. Let the set of coin denominations be $c_j, 0 \le j \le k$.
Consider an optimal solution for making change of i cents, and this optimal solution uses a coin with denomination c_j.
Similar to the proof in a. and b., we can prove that the optimal solution for $n-c_j$ cents will use $a-1$

coins, if the optimal solution for n uses a coins.
In other words, $number[i] = 1 + number[i - c_j]$.
Therefore, we can solve the *number* recursively by the algorithm:
Base case: $i = 0, number[i] = 0$.
Recursive step: $i \geq 1, number[i] = 1 + min_{0 \leq j \leq k}\{number[i - c_j]\}$.
Therefore, we first call the algorithm STORE_CHANGE(n, c, k), which will output two lists *number* and *coin*. We store in $number[i]$ the fewest number of coins we can use to make change for i cents. We store in $coin[i]$ the denomination of a coin used in an optimal solution for making change of i cents.
Then we can call another function MAKE_CHANGE$(n, coin)$ to output the coins used in the optimal solution for making change of n.

STORE_CHANGE(n, c, k)

```
0: for 1 ≤ i ≤ n do
0:     number[i] ← i
0:     for 0 ≤ j ≤ k do
0:         if i ≥ c_j and 1 + number[i − c_j] < number[i] then
0:             number[i] ← 1 + number[i − c_j]
0:             coin[i] ← c_j
0:         end if
0:     end for
0: end for=0
```

MAKE_CHANGE$(i, coin)$

```
0: if n > 0 then
0:     output coin[i]
0:     MAKE_CHANGE(i − coin[i], coin)
0: end if=0
```

The runtime for STORE_CHANGE(n, c, k) is $O(nk)$, and the runtime of MAKE_CHANGE$(n, coin)$ is $O(n)$. Therefore, the total runtim is $O(nk)$.