Homework 5

Anthony Weems

November 10, 2015

1. **Master Method**

   (a) $1 < \log_2 5 \implies$ T(n) $\in \Theta(n^2.3)$
   (b) $2 < \log_2 5 \implies$ T(n) $\in \Theta(n^2.3)$
   (c) $3 > \log_2 5 \implies$ T(n) $= \Theta(n^3)$

2. **Divide and Conquer**

   For simplicity, let us assume the sequences S_1 and S_2 are sorted from largest to smallest. This means, in a single sequence, the k^{th} largest element is simply the k^{th} element in the sequence. We will use this fact as the base case of our algorithm.

```
func(S1, S2, k)
  if S1 is empty return S2[k]
  if S2 is empty return S1[k]
  find the middle index of S1 and S2
  if middle_1 + middle_2 < k
    if S1[middle_1] > S2[middle_2]
      return func(S1, right half of S2, k - middle_2 - 1)
    else
      return func(right half of S1, S2, k - middle_1 - 1)
  else
    if S1[middle_1] > S2[middle_2]
      return func(left half of S1, S2, k)
    else
      return func(S1, left half of S2, k)
```

   This algorithm splits one of the sequences at each recursive step, eventually reaching the base case. Each step we reduce the problem to finding the $\frac{k}{2}^{th}$ largest element in the left or right half of the sequences.

3. **Divide and Conquer**

   We can devise a simple algorithm to by splitting the card set in half and checking the for equivalent cards in each half.

```
helper(S)
  n := size of S
  if n == 1 or (n == 2 and S[0] == S[1])
    return S[0]
  end

  S1 := left half of S
  S2 := right half of S

  C1 := helper(S1)
  C2 := helper(S2)

  if C1 != null
    check C1 against all cards in S
    if C1 matches at least n/2 cards
      return C1
    else
      check C2 against all cards in S
      if C2 matches at least n/2 cards
        return C2
      else
        return null

func(S)
  return helper(S) != null
```

   The helper function finds a card within S that matches at least $\frac{n}{2}$ cards by splitting the problem in two and searching each half for the same condition.

4. **Recurrences**

   The recurrence relation for the function foo is

$$T(n) = n + T(n/2) + T(n/2) + n/2 \tag{0.1}$$

   Using the master method, we can see $T(n) = \Theta(n \log n)$.

5. **Dynamic Programming**

   (a) $\Theta(c^n)$ for $c = 2$. The recursion tree has depth of n.

   (b) The given algorithm recomputes values at each step.

   (c) Asymptotic upper bound: $\Theta(n)$.

```
Fib(n)
  if n is 1 or 2, return 1
  if n in M, return M[n]
  M[n] = Fib(n-1) + Fib(n-2)
  return M[n]
end
```

   (d) Asymptotic upper bound: $\Theta(n)$.

```
Fib(n)
  F[0] = F[1] = 1
  for i in range(2, n)
    F[i] = F[i-2] + F[i-1]
  end
  return F[n]
end
```

6. **Dynamic Programming**

   We will create a matrix D such that $D[i, j]$ is true is some subset of elements up to i adds to j. Solving for $D[n, Z]$ will tell us the answer to the original question.

   For each entry, we will either include a_i in the subset or exclude it from the subset. Excluding it references $D[i-1, j]$ while including it references $D[i-1, j-a_i]$. This leads to the following equation:

$$D[i, j] = D[i-1, j-a_i] + D[i-1, j] \tag{0.2}$$

   Filling the table with traditional bottom-up dynamic programming requires examining each entry in the n-by-Z table, however, each entry is $O(1)$, therefore, the overall algorithm can be computed in $O(n * Z)$.