CMSC 142 Homework

Harold R. Mansilla (2015-46170)

December 17, 2018

Give an example of a problem that can be solved by the following algorithm design techniques.

1. Divide and conquer
2. Greedy
3. Dynamic Programming
4. Backtracking
5. Sieve
6. Branch and Bound
7. $\alpha - \beta$ Pruning

Indicate the input and the required output in each problem and the algorithm that solves the problem. Find the space and time complexity of the algorithm. Indicate the source of each (e.g. URL, bibliographic citation)

1 Divide and conquer

1.1 Problem

Merge Sort

1.2 Input

- arr - list to be sorted
- l - last index of array/sub-array to be sorted
- f - first index of array/sub-array to be sorted

1.3 Output

- arr - sorted list

1.4 Algorithm [1]

Algorithm 1 Merge Sort Algorithm

```
function MERGESORT(arr[n], l, r)
    int m = l+(r-l)/2;
    mergeSort(arr, l, m);
    mergeSort(arr, m+1, r);
    merge(arr, l, m, r);
end function
```

Algorithm 2 Merge Algorithm

```
function MERGE(A[n], l, m, r)
    int i, j, k
    int n1 = m - l + 1
    int n2 = r - m
    int L[n1], R[n2];
    for i ← 0 to n1 − 1 do
        L[i] = arr[l + i]
    end for
    for j ← 0 to n2 − 1 do
        R[j] = arr[m + 1+ j]
    end for
    i = 0;
    j = 0;
    k = l;
    while i < n1 and j < n2 do
        if L[i] <= R[j] then
            arr[k] = L[i]
            i++
        else
            arr[k] = R[j]
            j++
        end if
    end while
    while i < n1 do
        arr[k] = L[i]
        i++
        k++
    end while
    while j < n2 do
        arr[k] = R[j]
        j++
        k++
    end while
end function
```

1.5 Time Complexity

Let $T(n)$ be the time complexity of performing merge sort in a list of n elements.

$$
\begin{aligned}
T(n) &= \text{cost of sequence} \\
&= 2 + 3 + 4 + 5 \\
&= O(1) + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 5 \\
5 &= \text{cost of performing merge on array with } n \text{ elements} \\
5 &= O(n) \\
T(n) &= O(1) + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n) \\
T(n) &= 2T\left(\frac{n}{2}\right) + O(n) \\
\boxed{T(n) = O(nlgn)} & \text{ (by master method)}
\end{aligned}
$$

1.6 Space Complexity

Let $S(n)$ be the space complexity of performing merge sort

Variable	**Size**
A[n]	n
l	1
r	1
m	1

$$
\begin{aligned}
S(n) &= n + 1 + 1 + 1 \\
&= O(n) + O(1) + O(1) + O(1) \\
\boxed{S(n) = O(n)}
\end{aligned}
$$

2 Greedy

2.1 Problem

Coin Changing Problem

2.2 Input

- A - the amount to be converted
- D - array containing the denominations available

2.3 Output

- S - array containing the number of coins per denomination

2.4 Algorithm

Algorithm 3 Coin Changing Problem Algorithm

```
1:  function CCP(A, D)
2:      i = 1
3:      repeat
4:          curr = max(D)
5:          n_i = A div curr
6:          S = S ∪ n_i
7:          A = A − n_i curr
8:          D = D − {curr}
9:          i++
10:     until A == 0
11: end function
```

2.5 Time Complexity

Let $T(k)$ be the time complexity of the CCP algorithm with k denominations

$$
\begin{aligned}
T(k) &= \text{cost of sequence} \\
&= 2 + \text{3-10} \\
2 &= O(1) \\
\text{3-10} &= \text{number of iterations} * \text{cost of body of loop} \\
\text{cost of body of loop} &= 4 + 5 + 6 + 7 + 8 + 9 \\
&= O(k) + O(1) + O(1) + O(1) + O(1) + O(1) \\
\text{cost of body of loop} &= O(k)
\end{aligned}
$$

Note that the WCTC occurs when all denominations are used (i.e. by after kth denomination, A will then be equal to 0), therefore:

$$\boxed{T(k) = O(k)} \tag{1}$$

2.6 Space Complexity

Let $S(n)$ be the space complexity of solving the coin changing problem

Variable	**Size**
A	1
D	k
S	k
i	1
n_i	1
curr	1

$$\begin{aligned} S(n) &= 1 + k + k + 1 + 1 + 1 && (2) \\ &= O(1) + O(2k) + 1 + 1 + 1 && (3) \\ \boxed{S(n) = O(2k)} & && (4) \end{aligned}$$

3 Dynamic Programming

3.1 Problem

Newton Forward Difference Table

3.2 Input

- $x[m]$ - array containing x-values of the given m interpolating points (where $m = n + 1$)
- $y[m]$ - array containing y-values of the given m interpolating points (where $m = n + 1$)

3.3 Output

- $F[m][m]$ - forward difference table

3.4 Algorithm [2]

Algorithm 4 Newton Forward Difference Table Algorithm

```
1:  function NFD(x[m], y[m])
2:      F = Array[m][m]
3:      for i ← 0 to m − 1 do
4:          F[i][0] = y[i]                    ▷ Populate first column with the y-values
5:      end for
6:      for j ← 1 to m − 1 do
7:          for k ← 0 to m − j − 1 do
8:              F[k][j] = F[k + 1][j - 1] - F[k][j - 1]
9:          end for
10:     end for
11:     return F
12: end function
```

3.5 Time Complexity

Let $T(n)$ be the time complexity in computing the Newton Forward Difference table.

$$\begin{aligned} T(n) &= \text{cost of sequence} \\ &= 2 + \text{3-5} + \text{6-10} + 11 \end{aligned}$$

Computing at the cost of 2

$$2 = O(1)$$

Computing the cost of 3-5

$$\begin{aligned}
\text{3-5} &= \text{no. of iterations} * 4 \\
4 &= O(1) \\
\text{3-5} &= (m-1) * O(1) \\
&= O(m-1)
\end{aligned}$$

Computing the cost of 6-10

$$\begin{aligned}
\text{6-10} &= \text{no. of iterations} * \text{7-9} \\
\text{7-9} &= \text{no. of iterations} * 8
\end{aligned}$$

Note that at the maximum value of j, $m-j-1=0$ (which will get the value $\triangle^n f(a)$, or in this case: $\triangle^{m-1} f(a)$). The maximum value k can take is when $j=1$, or $max(k)=m-2$.

Therefore

$$\begin{aligned}
\text{7-9} &= (m-2) * 8 \\
&= (m-2) * (1) \\
&= O(m-2)
\end{aligned}$$

Now to get the cost of 6-10

$$\begin{aligned}
\text{6-10} &= \text{no. of iterations} * O(m-2) \\
&= (m-1) * O(m-2) \\
&= O(m^2-3m+2)
\end{aligned}$$

Computing the cost of 11

$$\text{11} = O(1)$$

Substituting the values into $T(n)$

$$T(n) = O(1) + O(m-1) + O(m^2-3m+2)$$
$$\boxed{T(n) = O(m^2)}$$

3.6 Space Complexity

Let $S(n)$ be the space complexity in computing the Newton Forward Difference table.

Variable	**Size**
x	m
y	m
F	m^2
i	1
j	1
k	1

$$
\begin{aligned}
S(n) &= m + m + m^2 + 1 + 1 + 1 \\
&= O(m) + O(m) + O(m^2) + O(1) + O(1) + O(1)
\end{aligned}
$$

$$\boxed{S(n) = O(m^2)}$$

4 Backtracking

4.1 Problem

N Queens

4.2 Input

- $board$ - the NxN chess board, assume to be initialized to all 0's
- col - the current column where the queen will be placed

4.3 Output

- $board$ - the board containing a solution to the N-Queens problem

4.4 Algorithm [3]

Algorithm 5 N Queens Algorithm

```
function NQ(board[N][N], col)
    if col >= N then
        return true
    end if
    for i ← 0 to N − 1 do
        if isSafe(board, i, col) then
            board[i][col] = 1
            if NQ(board, col + 1) then
                return true
            end if
            board[i][col] = 0
        end if
    end for
    return false
end function
```

Algorithm 6 Utility function to check if current queen placement can be attacked/can attack previously placed queens

```
function ISSAFE(board[N][N], row, col)
    int i, j
    for i ← 0 to col − 1 do                              ▷ Row on left side
        if board[row][i] == 1 then
            return false
        end if
    end for
    for i ← row, j ← col to 0,0 do                       ▷ Upper diagonal on left side
        if board[row][i] == 1 then
            return false
        end if
    end for
    for i ← row,j ← col to N − 1,0 do                    ▷ Lower diagonal on left side
        if board[row][i] == 1 then
            return false
        end if
    end for
end function
```

4.5 Time Complexity

Let $T(n)$ be the time complexity for solving the N Queens problem.

$$
\begin{aligned}
T(n) &= \text{cost of sequence} \\
&= \text{2-4} + \text{5-13} + \text{14}
\end{aligned}
$$

Computing the cost of 2-4

$$
\begin{aligned}
\text{2-4} &= max(\text{2}, \text{4}) \\
&= max(O(1), O(1)) \\
&= O(1)
\end{aligned}
$$

Computing the cost of 5-13

$$
\begin{aligned}
\text{5-13} &= \text{no. of iterations} * \text{6-12} \\
&= N * \text{6-12}
\end{aligned}
$$

Computing the cost of 6-12

$$
\text{6-12} = max(\text{6}, \text{7-11})
$$

Computing the cost of 6 is equivalent to finding the time complexity of the *isSafe* algorithm.

$$6 = O(N-1)$$

Computing the cost of 7-11

$$\begin{aligned}
\text{7-11} &= 7 + \text{8-10} + 11 \\
&= O(1) + max(8, 9) + O(1) \\
&= O(1) + max(T(N-i), O(1)) \\
&= O(1) + T(N-i) \\
&= T(N-i)
\end{aligned}$$

We can now get 6-12

$$\begin{aligned}
\text{6-12} &= max(O(N-1), T(N-i)) \\
&= T(N-i)
\end{aligned}$$

Now, we get 5-13

$$\text{5-13} = N * T(N-i)$$

Computing the cost of 14

$$14 = O(1)$$

We can now compute for $T(n)$

$$\begin{aligned}
T(n) &= O(1) + N * T(N-i) + O(1) \\
&= N * T(N-i)
\end{aligned}$$

Here, we note that the maximum value of $N-i$ (for the sake of computation) is $N-1$.

$$T(n) = N * T(N-1)$$
$$\boxed{T(n) = O(n^n)}$$

4.6 Space Complexity

Let $S(n)$ be the space complexity for solving the N Queens problem.

Variable	Size
board	N^2
col	1
i	1

$$
\begin{aligned}
S(n) &= N^2 + 1 + 1 \\
&= O(N^2) + O(1) + O(1)
\end{aligned}
$$

$$\boxed{S(n) = O(N^2)}$$

5 Sieve

5.1 Problem

Sieve of Eratosthenes

5.2 Input

- n - user-supplied input wherein the prime numbers will be computed from 0 to n

5.3 Output

- prime[n + 1] - boolean array containing the prime numbers from 0 to n

5.4 Algorithm [4]

Algorithm 7 Sieve of Eratosthenes Algorithm

```
function SIEVE(n)
    prime = Array[n + 1]
    for i ← 0 to n do
        prime[i] = true
    end for
    p = 2
    while p^2 <= n do
        if prime[p] == true then
            for i ← p^2 to n; i+ = p do
                prime[i] = false
            end for
        end if
        p + +
    end while
end function
```

5.5 Time Complexity

The inner loop does $dfracni$ steps, where i is prime. The whole complexity is $\sum \frac{n}{i} = n * \sum \frac{1}{i}$. According to prime harmonic series, $\sum 1/i$ where i is prime is $\log{(logn)}$. In total, $O(n \log{(\log n)})$.

$$\boxed{T(n) = O(n \log{(\log n)})}$$

5.6 Space Complexity

Let $S(n)$ be the space complexity of solving the Sieve of Eratosthenes algorithm.

Variable	**Size**
n	1
prime	n+1
p	1

$$\begin{aligned} S(n) &= 1 + (n+1) + 1 \\ &= O(1) + O(n+1) + O(1) \\ &= O(n+1) \end{aligned}$$

$$\boxed{S(n) = O(n)}$$

6 Branch and Bound

6.1 Problem

$0/1$ Knapsack problem

6.2 Input

- W - the weight capacity of the knapsack
- arr[] - array containing the items. Each element of the array consists of the price/value and its weight
- n - size of the array

6.3 Output

- maxProfit - the maximum profit that can be derived from the items

6.4 Algorithm [5]

Algorithm 8 Comparison function to sort items by value/weight ratio

```
function CMP(a, b)
    r1 = a.value / a.weight;
    r2 = b.value / b.weight;
    return r1 > r2;
end function
```

Algorithm 9 Bound function to compute the maximum profit bound of a subtree rooted in u

```
function BOUND(u, n, W, arr[])
    if u.weight >= W then
        return 0
    end if
    profit-bound = u.profit
    j = u.level + 1
    totweight = u.weight
    while j < n and totweight + arr[j].weight ¡= W do
        totweight += arr[j].weight
        profit-bound += arr[j].value
        j++
    end while
    if j < n then
        profit-bound += (W - totweight) * arr[j].value / arr[j].weight
    end if
    return profit-bound
end function
```

Algorithm 10 0/1 Knapsack Problem Algorithm

```
1: function KNAPSACK(W, arr[], n)
2:     sort(arr, arr + n, cmp)
3:     queue<Node>Q
4:     Node u, v
5:     u.level = -1
6:     u.profit = u.weight = 0
7:     Q.enqueue(u)
8:     maxProfit = 0
9:     while !Q.empty() do
10:        u = Q.front()
11:        Q.dequeue()
12:        if u.level == -1 then
13:            v.level = 0
14:        end if
15:        if u.level == n-1 then
16:            continue
17:        end if
18:        v.level = u.level + 1
19:        v.weight = u.weight + arr[v.level].weight
20:        v.profit = u.profit + arr[v.level].value
21:        if v.weight <= W and v.profit > maxProfit then
22:            maxProfit = v.profit
23:        end if
24:        v.bound = bound(v,n,W,arr)
25:        if v.bound > maxProfit then
26:            Q.enqueue(v)
27:        end if
28:        v.weight = u.weight
29:        v.profit = u.profit
30:        v.bound = bound(v,n,W,arr)
31:        if v.bound > maxProfit then
32:            Q.push(v)
33:        end if
34:    end while
35: end function
```

6.5 Time Complexity

Let $T(n)$ be the time complexity of solving the 0/1 Knapsack problem using branch and bound. We assume that $T(n) =$ 9-34 as the preceding statements only take constant time except for the sorting of the array which depends on the sorting algorithm used as well as the comparison function.

$$\boxed{T(n) = O(2^n)}$$

6.6 Space Complexity

Let $S(n)$ be the space complexity of solving the 0/1 Knapsack problem using branch and bound.

Variable	**Size**
W	1
arr	n
Q	n
u	1
v	1

$$S(n) = 1 + n + n + 1 + 1$$
$$\boxed{S(n) = O(n)}$$

7 $\alpha - \beta$ Pruning

7.1 Problem

Minimax Algorithm

7.2 Input

- node - current node in the game tree
- depth - current depth in the game tree
- isMaximizingPlayer - boolean value that verifies whether the player is the maximizing player or otherwise
- α - the best value the *maximizer* can have at a node level in the game tree
- β - the best value the *minimizer* can have at a node level in the game tree

7.3 Output

- bestVal - the best value to be taken by the player at the call of the function

7.4 Algorithm [6]

Algorithm 11 $\alpha - \beta$ pruning technique for the Minimax algorithm

1: **function** MINIMAX(node, depth, isMaximizingPlayer, α, β)
2: **if** node is a leaf node **then**
3: **return** node.value
4: **end if**
5: **if** isMaximizingPlayer **then**
6: bestVal = $-\infty$
7: **for each** child node **do**
8: value = minimax(node, depth + 1, false, α, β)
9: bestVal = max(bestVal, value)
10: alpha = max(alpha, bestVal)
11: **if** $\beta <= \alpha$ **then**
12: **break**
13: **end if**
14: **end for**
15: **return** bestVal
16: **else**
17: bestVal = ∞
18: **for each** child node **do**
19: value = minimax(node, depth + 1, true, α, β)
20: bestVal = min(bestVal, value)
21: beta = min(alpha, bestVal)
22: **if** $\beta <= \alpha$ **then**
23: **break**
24: **end if**
25: **end for**
26: **return** bestVal
27: **end if**
28: **end function**

7.5 Time Complexity [7]

Let $T(n)$ be the time complexity for using the $\alpha - \beta$ pruning technique for the minimax algorithm. Let b be the number of child nodes of a node. Let d be the depth of the tree. We assume that all non-leaf nodes have b children.

$$\begin{aligned} T(n) &= \text{cost of sequence} \\ &= \text{2-4} + \text{5-27} \end{aligned}$$

Computing for the cost of 2-4

$$\begin{aligned} \text{2-4} &= max(2, 3) \\ &= max(O(1), O(1)) \\ \text{2-4} &= O(1) \end{aligned}$$

Variable	Size
node	bd
depth	1
isMaximizingPlayer	1
α	1
β	1
bestVal	1

Computing the cost of 5-27

$$\text{5-27} = max(5, \text{6-15}, \text{17-26})$$

Note here that the costs of 6-15 and 17-26 are equal assuming that we are considering the same initial game tree (at the first call of the function) and the only difference is whether the player is the maximizing or minimizing player.

$$\begin{aligned}\text{6-15} &= 6 + \text{7-14} + 15 = \text{17-26} \\ &= O(1) + \text{no. of iterations} * \text{8-13} + O(1) \\ &= O(1) + b * \text{8-13} + O(1)\end{aligned}$$

Computing the cost of 8-13

$$\begin{aligned}\text{8-13} &= 8 + 9 + 10 + max(11, 12) \\ &= T() + O(1) + O(1) + max(O(1), O(1)) \\ &= T() + O(1) + O(1) + O(1)\end{aligned}$$

$$\boxed{T(n) = O(b^{\frac{d}{2}})}$$

7.6 Space Complexity [7]

Let $S(n)$ be the space complexity for using the $\alpha - \beta$ pruning technique in the Minimax algorithm. Let b be the number of child nodes of a node. Let d be the depth of the tree.

$$S(n) = bd + 1 + 1 + 1 + 1 + 1 \quad (5)$$

$$= O(bd) + O(1) + O(1) + O(1) + O(1) + O(1) \quad (6)$$

$$\boxed{S(n) = O(bd)} \quad (7)$$

Bibliography

[1] *Merge sort*. [Online]. Available: https://www.geeksforgeeks.org/merge-sort/ (visited on 12/16/2018).

[2] *Newton forward and backward interpolation*. [Online]. Available: https://www.geeksforgeeks.org/newton-forward-backward-interpolation/ (visited on 12/16/2018).

[3] *N queen problem — backtracking 3*. [Online]. Available: https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/ (visited on 12/16/2018).

[4] *Sieve of eratosthenes*. [Online]. Available: https://www.geeksforgeeks.org/sieve-of-eratosthenes/ (visited on 12/17/2018).

[5] *Implementation of 0/1 knapsack using branch and bound*. [Online]. Available: https://www.geeksforgeeks.org/implementation-of-0-1-knapsack-using-branch-and-bound/ (visited on 12/17/2018).

[6] *Minimax algorithm in game theory — set 4 (alpha-beta pruning)*. [Online]. Available: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/ (visited on 12/17/2018).

[7] *Cis603 s03*. [Online]. Available: https://cis.temple.edu/~vasilis/Courses/CIS603/Lectures/l7.html (visited on 12/17/2018).