WEEK5: BACKTRACKING, GREEDY ALGORITHMS & MIDTERM

Agenda

Survey Comments

MidTerm preparation

Time complexity

n Exploration: Backtracking - Combination Sum Problem: "Problem: Given a sorted array of positive integers nums[] and a sum x, find all unique combinations of integers from the nums[] array whose sum is equal to x. Any integer in the array can be chosen an unlimited number of times. " Time complexity is exponential. If n is the length of the array in the question and k is the target sum. Time complexity will be O(n^k). Please explain why the time complexity is n^k

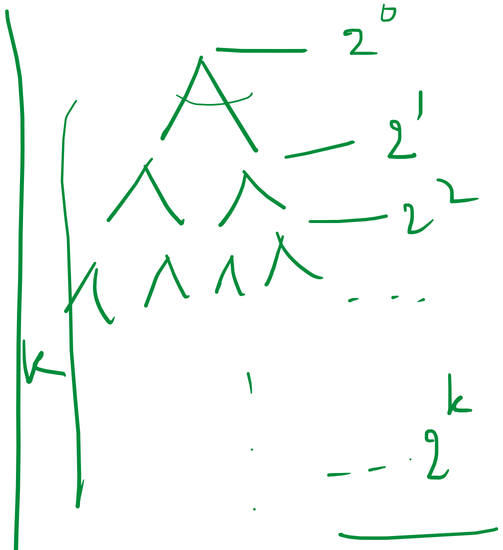
Time complexity- combination sum

[1, 2, 3, 4] n , k

Repeats

(1) K

$(n)^k$



$O(2^k)$

(2)   result [( ) _ _ _ _ .]

Total number of combinations

n choose k = ${}^nC_k$

$$\frac{n!}{k!(n-k)!} = \frac{n\cdot(n-1)(n-2)\cdots\cancel{(n-k)!}}{k!\ \cancel{(n-k)!}} = \frac{\overbrace{n\cdot(n-1)(n-2)\cdots(n-k+1)}^{k \text{ terms}}}{k!} \approx O(n^k)$$

Time complexity

```python
def is_attacked(row, col, board, N):
    #check row
    for i in range(N):
        if(board[row][i] == 1):
            return True

    #check column
    for i in range(N):
        if(board[i][col] == 1):
            return True

    #check upper left diagonal cells
    row_p = row-1
    col_p = col-1
    while(row_p>=0 and col_p>=0):
        if(board[row_p][col_p] == 1):
            return True
        row_p -=1
        col_p -=1
    #check upper right diagonal cells
    row_p = row-1
    col_p = col +1
    while(row_p>=0 and col_p<N):
        if(board[row_p][col_p] ==1):
            return True
        row_p -=1
        col_p +=1

    return False
```

```python
def solve_n_Queens(board,row, N, remaining):
    #base case if solved for N rows return
    if(remaining==0):
        return True

    for col in range(N):
        if(is_attacked(row, col, board, N)):
            continue        #skip the attacked cell
        else:
            board[row][col] = 1
            if(solve_n_Queens(board, row+1,N, remaining-1)): # recursively solve for solution
                return True
            #backtrack if any placement results in no solution
            board[row][col]=0
    return False
```

$O(n^2)$

$nT(n-1)$

```python
def n_Queens(N):
    board = [[0 for x in range(N)] for x in range(N)]

    solve_n_Queens(board,0, N, N)
    return board

print(n_Queens(4))
```

$$T(n) = nT(n-1) + n^2$$

$$= n\{(n-1)T(n-2) + (n-1)^2\} + n^2$$

$$= n(n-1)T(n-2) + n(n-1)^2 + n^2$$

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

$$= n(n-1)(n-2)T(n-3) + n(n-1)(n-2)^2 + \cdot + n$$

$$\vdots$$

$$n(n-1)(n-2)(n-3)\cdots \underbrace{T(n-k)}_{c} + n$$

$$n!$$

Solving Time Complexity for Backtracking

→ (1) # of possible solutions → calculate this

(2) Analyse code – if possible.

Backtracking

Is backtracking essentially DFS? Except instead of visiting all nodes, you have constraints that will stop from continuing on a path if we know that the solution is not the be found down it? Is there a case where backtracking wouldn't be DFS?

- DFS uses the concept of backtracking

Mid term practice:

Comparing Function Growth

1. $f(n) = 0.01n^3$; $g(n) = 50n+10$

$f(n)$ vs $g(n)$.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

$$f(n) = \Omega\, g(n) \quad (\text{or}) \quad g(n) = O(f(n))$$

2. $f(n) = \log n^2$; $g(n) = \log n + 10$

$$f(n) = \log n^2 = 2 \log n$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \frac{2 \log n}{\log n + 10} = 2 \Rightarrow f(n) = \Theta(g(n))$$

$$\lim_{n \to \infty} \frac{T(n)}{g(n)} = \begin{cases} 0 & \text{implies that } T(n) \text{ has a smaller order of growth than } g(n). \quad T(n) \in O(g(n)) \\ c & \text{implies that } T(n) \text{ has the same order of growth as } g(n). \quad T(n) \in \theta(g(n)) \\ \infty & \text{implies that } T(n) \text{ has a larger order of growth than } g(n). \quad T(n) \in \Omega(g(n)) \end{cases}$$

Comparing Function Growth

4. $f(n) = \log n^3$; $g(n) = \log^3 n$

$f(n) = 3 \log n$; $g(n) = (\log n)^3$

$f(n) = O(g(n))$

$$\lim_{n \to \infty} \frac{T(n)}{g(n)} = \begin{cases} 0 & \text{implies that } T(n) \text{ has a smaller order of growth than } g(n). \quad T(n) \in O(g(n)) \\ C & \text{implies that } T(n) \text{ has the same order of growth as } g(n). \quad T(n) \in \theta(g(n)) \\ \infty & \text{implies that } T(n) \text{ has a larger order of growth than } g(n). \quad T(n) \in \boldsymbol{\Omega}(g(n)) \end{cases}$$

Comparing Function Growth

5. $f(n) = 10$; $g(n) = \log 10$

Both are constants

$f(n) = \Theta(g(n))$

6. $f(n) = 2^n$; $g(n) = 10n^2$

$$\lim_{n\to\infty} \frac{2^n}{10n^2} = \lim_{n\to\infty} \frac{(\log 2) \times (2^n)}{20n}$$ L'Hospital

$$= \lim_{n\to\infty} \frac{(\log 2)^2 \times (2^n)}{20}$$ L'Hosp

$$= \infty$$

$f(n) = \Omega(g(n))$

$$\lim_{n\to\infty} \frac{T(n)}{g(n)} = \begin{cases} 0 & \text{implies that } T(n) \text{ has a smaller order of growth than } g(n). \quad T(n) \in O(g(n)) \\ C & \text{implies that } T(n) \text{ has the same order of growth as } g(n). \quad T(n) \in \theta(g(n)) \\ \infty & \text{implies that } T(n) \text{ has a larger order of growth than } g(n). \quad T(n) \in \Omega(g(n)) \end{cases}$$

Analyze time complexity

1.

$j \leftarrow 0$
while $(j < n)$ **do**
  $j \leftarrow j + 2$
  $z \leftarrow z + 1$

AN ANSWER. Since j goes through the values 0, 2, 4, 6, ... until j reaches n (if n is even) or $n+1$ (if n is odd), the while loop goes through at most $\lceil n/2 \rceil$ many iterations. Hence, z is in $\Theta(n)$.

$\Theta(n)$

Analyze time complexity

2.

```
for k ← 0 to n do
   for j ← 0 to k do
      z ← z + 1
```

$\Theta(n^2)$

AN ANSWER. *Inner loop:* Since j goes from 0 to k, the inner loop has $(k+1)$-many iterations. So, z is increased by $(k+1)$. *Outer loop:* k goes from 0 to n. So z is increased by

$$\sum_{k=0}^{n}(k+1) = 1+2+\cdots+n+(n+1) = \frac{(n+1)(n+2)}{2} \in \Theta(n^2).$$

k: 0 1 2 n

j: 0 0,1 0,1,2 0,1,2..

#times 1 2 3 + .. n

Σn

Analyze time complexity

3.

$i \leftarrow n$
while $(i > 1)$ **do**
  $i \leftarrow \lfloor i/2 \rfloor$
  $z \leftarrow z + 1$

$\Theta(\log n)$

AN ANSWER. Since i takes on the sequence of values $n = n/2^0$, $\lfloor n/2 \rfloor = \lfloor n/2^1 \rfloor$, $\lfloor n/4 \rfloor = \lfloor n/2^2 \rfloor, \ldots, \lfloor n/2^j \rfloor$ until i is ≤ 1. The smallest value of i such that $n/2^i \leq 1$ is $\lceil \log_2 n \rceil$. So there are $(1 + \lceil \log_2 n \rceil)$-many iterations and $z \in \Theta(\log_2 n)$.

i: n, $n/2$, $n/2^2$ … $\frac{n}{2^k} = 1$

times 1 1 1 1

1^{st} 2^{nd} 3^{rd} k^{th}

$\frac{n}{2^k} = 1$

$\log n = k$

Analyze time complexity

4.

```
int sum = 0;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            if (i == j == k) {
                for (l = 0; l < n*n*n; l++) {
                    sum = i + j + k + l;
                }
            }
        }
    }
}
```

$i = 0, 1, 2, 3 \cdots n$

$j = 0, 1, 2$

$k = 0 \cdots n, 1, 2$

$\overbrace{1 + 1 + 1 + 1 - \cdots + 1}^{n}$

n^3

$\underbrace{n^3 + n^3 + \cdots n^3}_{n}$

$= n(n^3) = n^4$

$\underbrace{\sum_{i=0}^{n} \sum_{j=0}^{n} \sum_{k=0}^{n}}_{} \left[\left(\sum_{l=0}^{n^3} 1 \right) \right]_{ij}$

n^3

Analyze time complexity

5.

```
int count = 0;
for (int i = N; i > 1; i = i/2)
    for (int j = 0; j < i; j++)
        count++;
```

A $O(n^2)$

B $O(\log n)$

C $O(n)$

D None.

$i = N$ $\quad$ $N/2$ $\quad$ $N/2^2$ $\quad \ldots$ $\quad \frac{N}{2^k} = 1$

$j = 0 - N$ $\quad$ $0 - N/2$ $\quad$ $0 \ldots N/2^2$ $\quad$ $0 \ldots N/2^k$

N times $\quad$ $N/2$ times $\quad$ $\frac{N}{2^2}$ times $\quad$ $\frac{N}{2^k}$ times

$$= N + \frac{N}{2} + \frac{N}{2^2} + \ldots \frac{N}{2^k}$$

G.P.

$O(n)$

Recurrence Relation & Recurrence Formula

```
Fib(n):
   if n = 0:
      return 1
   else if n = 1:
      return 1
   else:
      return Fib(n-1) + Fib(n-2)
```

Recurrence relation

T(n) = c1 for n<=1

T(n) = T(n-1) +T(n-2) + c2 otherwise

Recurrence Formula

$$F(n) = \begin{cases} 1 & \text{If n = 0} \\ 1 & \text{If n = 1} \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

Recurrence Relation & Recurrence Formula

```
def Find_Max_Array(Arr,n):
   if(n==1) then
      return(Arr[1])

   else
      return(max(Arr[n],Find_Max_Array(Arr,n-1)))
```

Recurrence relation

$$T(n) = c1 \qquad n=1$$

$$T(n) = T(n-1) + c2 \quad \text{otherwise}$$

Recurrence Formula

$$F(1) = A[1]$$

$$F(n) = \max(A[n], F(n-1))$$

Go Through the Recurrence Formulas for the problems covered in the explorations.

Midterm preparation Strategy

The algorithm strategies covered in explorations.

- What?
- When to use them?
- How do the problems in the explorations use them.

Interactive activities that are part of the explorations.

Are you able to answer the questions given in the overview page of each week.

Can you solve the problems given in the explorations?

Sample Question

Given two numbers n, k, and find an algorithm that outputs all combinations of k numbers in [1..n]

Which of the following techniques could be used?

A. Greedy Approach
B. Divide and Conquer
C. Backtracking
D. Dynamic Programming

Sample Question

In molecular biology, DNAs and proteins can be represented as a sequence of alphabets. DNA sequences consist of A, T, G, C representing nucleobases adenine, thymine, guanine and cytosine.

Two sample DNA sequences GACGGATTAG and GATCGGAATAG.

You are given two sequences query sequence and database sequence. You need to find the similarity between them. Similarity between sequences is the longest matching subsequence.

This problem is similar to which of the problems from the explorations?

A. Knapsack
B. N-Queens
C. Longest Common Subsequence
D. Change Making Problem

Can you write its recurrence formula?
Can you write pseudocode to solve this problem