CSL 3010: Design and Analysis of Algorithms July-November 2022

Lecture 8: Recursion and Backtracking

Lecturer: Dr. Saket Saurabh *Scribe: Mukul, Sawan, Jaimin, Tanay*

1 Subset Sum

Input : Given a set X of positive integers and a target T.
Output : To check whether there is a subset of elements in X that adds up to T.

Example case 1: X = [8, 6, 7, 5, 3, 10, 9] and T = 15.
O/P: **Yes, subset exists**. Following are some of the subsets, sum of whose elements equals the required target [[5, 10], [7, 8], [5, 3, 7], [9, 6]].

Example case 2: X = [11, 6, 5, 1, 7, 13, 12] and T = 15.
O/P: **No subset exists**.

Approach : We will appoach this via a **Recursive approach** and generally it comprises of the following two things :-

- Base case Handling
- General case Handling

suppose there exists an element $x \in X$, it will have only two choices :-

- There is a subset of X that **includes** x and sums to T. This will further imply that there exists a subset X-{x} that sums to T-x. so the recursive call will be made as per those parameters only.
- There is a subset X that **does not** include x and sums to T. This will further imply that there exists, a subset X-{x} that sums to T. so the recursive call will be made as per those parameters only.

Base cases :-

- T = 0, return True.
- T <0 or X = {}, return False.

Pseudo code:

```
def SUBSET_SUM(X,T):

    If T == 0:
        return True
    elif T < 0 or X = {}:
        return False
    else:
        let x be an element of X
        taken = SUBSET_SUM(X-{x},T-x)
        not_taken = SUBSET_SUM(X-{x},T)
        return (taken V not_taken)
```

Listing 1: Pseudo code for subset sum

Proof of correctness :-
We'll prove this algorithm is correct by induction on $|X|$.

- If T = 0, then {} empty subset sum is 0 so output is True.
- if T < 0 or X = {}, since all numbers are positive then they can't make up to a negative sum also, no subset of X can sum up to T, so output is False.
- otherwise, if there exists a subset of X that sums up to T, either it contains x or it doesn't, the Recursion fairly correctly checks for each of those possibilities and induction takes care of both cases since $|X - \{x\}| < |X|$.

1.1 Analysis

In order to properly analyze the algorithm, we have to be a bit more precise about the implementation. Let's assume that the input sequence X is given as an array X[1 .. n]. We pass only three values as arguments the starting address of the subarray , index i and the target T. Alternatively, we could avoid passing the same starting address X to every recursive call by making X a global variable

```
Subset_Sum(X, i, T):
    if T = 0:
        return True
    else if T < 0 or i = 0 :
        return False
    else
        taken = Subset_Sum(X, i-1, T-X[i])
        not_taken =  Subset_Sum(X, i-1, T)
    return (taken V not_taken)
```

Listing 2: Pseudo code II for subset sum

The running time T(n) of our algorithm satisfies the recurrence T(n)≤ 2T(n - 1) + O(1). This will finally evaluate to $\mathbf{O(2^n)}$.
In the worst case will be when T is lager than sum of all elements of X - recursion tree will be complete

2 Text Segmentation

Input : string of letters
Output : If a string segmentation exists such that each word is an English Language word then return the segmented string.

Example case : " BOTHEARTHANDSATURNSPIN "
O/P : {" BOTH EARTH AND SATURN SPIN ", " BOT HEART HAND SATURN SPIN ", " BOT HEART HANDS AT URNS PIN " }

Approach : let's assume that we have access to a Function **IsWord(w)** that takes a string w as input, returns True if w is a word, and returns False if w is not a word.
Main thing is to decide where the next word in the output sequence ends. Suppose we have iterated till BOTH in the string so for the remaining part there are four possibilities for the next output word:
{HE, HEAR, HEART, HEARTH}.
We don't know which of these choices, if any, is consistent with a complete segmentation of the input string. Our backtracking algorithm "stupidly" tries every possibility by brute force, and lets the Recursion do all the real work.

We will follow a **natural backtracking strategy** i.e. Select the first output word, and recursively segment the rest of the input string.
To get a complete recursive algorithm, we need a base case. Our recursion will stop when we reach the end of the input string, because there is no next word. And we know, the empty string has a unique segmentation into zero words! Concluding all of this, we arrive at the following recursive algorithm:

```
Splittable(A[1 .. n]):
    if n = 0:
        return True
    for i = 1 to n :
        if IsWord(A[1 .. i]):
            if Splittable(A[i + 1 ... n]):
                return True
    return False
```

Listing 3: Pseudo code for text segmentation

2.1 Improvement

It's seen that passing arrays as input parameters to algorithm is rather slow, instead, it's incredibly useful to treat the original input array as a global variable and then reformulate the problem and the algorithm in terms of array indices instead of explicit subarrays. For our string segmentation problem, the argument of any recursive call is always a suffix A[i...n] of the original input array. So we can modify our recursive algorithm like this using these two boolean functions:

- For any indices i and j, we define IsWord(i, j) = True iff the substring A[i ... j] is a word.
- For any index i, we define Splittable(i) = True iff the suffix A[i ... n] can be split into words.

For example, IsWord(1, n) = True if and only if the entire input string is a single word, and Splittable(1) = True if and only if the entire input string can be segmented. Our recursive strategy gives us the following recurrence:

$$\text{splittable(i)} = \begin{cases} TRUE & if,\ i > n \\ \vee_{j=i}^{n}\ IsWord(i,j) \wedge splittable(j+1) & otherwise \end{cases}$$

Pseudocode:

```
# This checks whether the suffix A[i .. n] Splittable ?
Splittable(i):
    if i > n:
        return True
    for j = i to n:
        if IsWord(i, j):
            if Splittable(j + 1):
                return True
    return False
```

Listing 4: Pseudo code II for text segmentation

Although it may look like a small difference, using index notation instead of array notation is an important habit, to speed up backtracking algorithms.

2.2 Analysis

IsWord runs in O(1) time. Our algorithm calls IsWord on every prefix of the input string, and possibly calls itself recursively on every suffix of the output string. Thus, the running time of Splittable is determined b this recurrence realtion.

$$\begin{gathered} \text{T(n)} \leq O(n) + \sum_{i=0}^{n-1} T(i) \\ \text{T(n)} = \alpha(n) + \sum_{i=0}^{n-1} T(i) \\ \text{T(n)} = \alpha(n-1) + \sum_{i=0}^{n-2} T(i) \\ \text{T(n) - T(n-1)} = \alpha + T(n-1) \\ \text{T(n)} = 2\text{T(n-2)} + \alpha \end{gathered}$$

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

There are exactly 2 power n-1 possible ways to segment a string of length n, In the worst case, the algorithm explores each of these $2^{n1} possibilities$.

3 Longest Increasing Sub-sequence

Input : sequence of integers.
Output : find the length of the longest subsequence whose elements are in increasing order.

For any sequence S, a **subsequence** of S is another sequence from S obtained by deleting zero or more elements, without changing the order of the remaining elements; the elements of the subsequence need not be together in the original sequence S.

What we need to do ?
we are given a sequence of integers, and we need to find the longest subsequence whose elements are in increasing order. More concretely, the input is an integer array A[1 ... n], and we need to compute the longest possible sequence of indices 1 <= $i_1 < i_2 <...< i_l \leq nsuchthat A[i_k] < A[i_{k+1}] forallk$.

3.1 Approach

Brute force approach for this problem is to decide, for each index j in order from 1 to n, whether or not to include A[j] in the subsequence. Let's look at this example:

3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 7

Let's say the green highlighted part depicts the part of the sequence till which we have iterated, say we decided to include {1, 4, 5, 6} this subsequence towards our answer and ignore all other iterated numbers, then, we'll check fo further elements whether we can take them or not.
So we cannot include 5, because then the selected numbers would no longer be in increasing order. So let's skip ahead to the next decision for element 8

3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 7

Now we can include 8, but it's not obvious whether we should. Rather than trying to be clever , our backtracking algorithm will use simple brute force.

- First tentatively include the 8, and let the Recursion make the rest of the decisions.
- Then tentatively exclude element 8, and let the Recursion make the rest of the decisions.

Whichever choice leads to a longer increasing subsequence is the right one.

So **what we really need to solve is** :-

Given an integer prev and an array A[1 ... n], find the longest increasing subsequence of A in which every element is larger than prev.

Base case :
This algorithm has edge case when we get to the end of the array, because there is no next number to consider. But an empty array has exactly one subsequence, namely, the empty sequence. Considering, every element in the empty sequence is larger than whatever value we want, and every pair of elements in the empty sequence appears in increasing order. Thus, the longest increasing subsequence of the empty array has length 0.

Let's take a look at the recursive algorithm for the same.

```
LISbigger(prev,A[1 ... n]):
    if n = 0:
        return 0
    else if A[1] <= prev:
        return LISbigger(prev,A[2 .. n])
    else:
        taken = 1 + LISbigger(A[1],A[2 ... n])
        not_taken = LISbigger(prev,A[2 ... n])
    return max{taken, not_taken}
```

Listing 5: Pseudo code for Longest Increasing Subsequence

3.2 Improvement

Passing arrays around on the call stack is expensive; let's try to rephrase everything in terms of array indices, assuming that the array A[1 ... n] is a global variable. The integer prev is typically an array element A[i], and the remaining array is always a suffix A[j ... n] of the original input array. So we can **reformulate our recursive problem as follows:**

Given two indices i and j, where i < j, find the longest increasing subsequence of A[j .. n] in which every element is larger than A[i]

Let LISbigger(i, j) denote the length of the longest increasing subsequence of A[j ... n] in which every element is larger than A[i]. Our recursive strategy gives us the following recurrence

$$\text{LISbigger(i, j)} = \begin{cases} 0 & if \;\; j > n \\ LISbigger(i, j+1) & if A[i] > A[j] \\ max\{1 + LISbigger(j, j+1), LISbigger(i, j+1)\} & otherwise \end{cases}$$

Pseudocode:

```

LISbigger(i, j):
    if n = 0:
        return 0
    else if A[1]     prev:
        return LISbigger(i, j + 1)
    else:
        taken = 1 + LISbigger(j, j + 1)
        not_taken =  LISbigger(i, j + 1)
    return max{taken, not_taken}
```

Listing 6: Pseudo code II for Longest Increasing Subsequence

Finally, we need to connect our recursive strategy to the original problem: Finding the longest increasing subsequence of an array with no other constraints. we can add an artificial sentinel value in the beginning of the array i.e. -∞.
so, $A[0] = -\infty$

3.3 Analysis

In the worst case, the algorithm examines each of the subsequences of the input array. The running time of LISbigger satisfies the recurrence T(n) <= 2T(n-1)+O(1), which as usual implies that $\mathbf{T(n) = O(2^n)}$.