1 Introduction

Solving any problem on a computer, in the end boils down to a reduction...

1. C program → assembly code.
2. Using reductions enable us to work in an abstract/high level.

2 Subset Sum

Consider the subset sum problem:

**Input:** A set of \( n \) positive integers, and integer \( T \).

**Question:** Compute a subset \( Y \subseteq X \) such that \( \sum_{y \in Y} y = T \).

Example: \( X = \{2,5,8,11\}, T = 18 \).

Solution:

\[ Y = \{2,5,11\} \text{ since } 2+5+11 = 18. \]

Assume we have a black box \( ExistsSubsetSum(X,T) \) which returns true, if and only if there is a subset of \( X \) which sums up to \( T \).

**Question:** Can one compute efficiently the subset \( Y \) realizing \( SubsetSum(X,T) \) using the black box \( (ExistsSubsetSum) \)?

Idea: Consider an element \( \alpha \in X \) either it appears in all subsets that sums to \( T \) or not. We can check this by asking \( ExistsSubsetSum(X \setminus \{\alpha\}, T) \).

If \( ExistsSubsetSum(X,T) \) is false, then we are done.

Indeed, if \( ExistsSubsetSum(X \setminus \{\alpha\}, T) \) is true, then we don’t need \( \alpha \) at all. If it is false, then call \( ExistsSubsetSum(X \setminus \{\alpha\}, T - \alpha) \) [which must be true BTW] and write down \( \alpha \) as appearing in the output.
ConstructSubset(\(X[1..n], T\))
If \(T = 0\) Then return \(\emptyset\)
If \(X = \emptyset\) Then return NULL
If (not ExistsSubsetSum(\(X, T\)) Then return NULL.
\(Y \leftarrow \emptyset\), \(j \leftarrow 1\)
For \(i = 1, \ldots, n\) do
    \(\alpha \leftarrow X[i]\)
    If ExistsSubsetSum(\(X[i+1..n], T\)) Then
        continue;
    If ExistsSubsetSum(\(X[i+1..n], T - \alpha\)) Then
        \(Y[j] = \alpha;\)
        \(j \leftarrow j + 1;\)
        \(T \leftarrow T - \alpha.\)
return \(Y.\)

**2.1 Thinking Recursively**

So, how do we solve a problem recursively?

1. Check boundary cases and handle them easily.

2. Reduce the given instance into several smaller instances.

3. Solve the smaller instances recursively
   (Think about a recursion fairy that solves the instances for you magically.)

4. Merge the solution of the subproblems into a solution to the given instance.

Divide and conquer.
2.2 Running time

What's the running time of the algorithm ConstructSubset, if we assume calling ExistsSubset takes constant time (i.e., $O(1)$)?

Well, let's write a recursive formula for the running time

$$T(n) = T(n - 1) + O(1)$$

Solution to this recurrence?

A: $O(n)$.

2.3 From ConstructSubset to ExistsSubsetSum

Assume that we are now given ConstructSubset as a black box, and we want to use it to build a function ExistsSubsetSum. Of course, we can do the trivial thing... But let us instead use the same logic we did in constructing ConstructSubset. We get:

```
ExistsSubsetSum(X[1..n], T)
  If T = 0 Then
    return True
  If X = ∅ or T < 0 Then
    return False
  α ← X[1]
  Y ← ConstructSubset(X[2..n], T)
  If Y ≠ NULL Then
    return True
  Y ← ConstructSubset(X[2..n], T - α)
  If Y ≠ NULL Then
    return True
  return False.
```

- ExistsSubsetSum calls ConstructSubset on smaller instances
- ConstructSubset calls ExistsSubsetSum also on smaller instances.
- Thus, we are done!
- We have a real algorithm that can solve the SubsetSum problem!

This is of course, silly, to some extent, because we can solve ExistsSubsetSum directly:

```
ExistsSubsetSum(X[1..n], T)
  If T = 0 Then
    return True
  If X = ∅ or T < 0 Then
    return False
  return ExistsSubsetSum(X[2..n], T - α)
```

\[ \checkmark \]
And that’s cleaner...

Running time:

\[ T(n) = 2T(n - 1) + O(1) \]

Which is

\[ T(n) = 2T(n - 1) + 1 = 2(2T(n - 2) + 1) + 1 \]
\[ = 4T(n - 2) + 3 = 2^2T(n - i) + 2^i - 1 \]
\[ = 2^{n-1}T(1) + 2^{n-1} - 1 \]
\[ = O(2^{n-1}) = O(2^n). \]

Final word, we can do ConstructSubset also directly.

```
ConstructSubset(X[1..n], T)
If T = 0 Then
   return \emptyset
(* X = \emptyset can be checked by checking n = 0 *)
If n = 0 or T < 0 Then
   return NULL
Y \leftarrow ConstructSubset(X[2..n], T)
If Y \neq NULL then
   return Y
Y \leftarrow ConstructSubset(X[2..n], T - X[1])
If Y \neq NULL then
   return Y
return NULL
```

The running time of this algorithm is also \( O(2^n) \). Since this problem is NP-Complete this is close to the best one can hope for.

3 Longest Increasing Subsequence

Suppose we want to find the longest increasing subsequence of a sequence of \( n \) integers.

**Input:** array \( A[1..n] \) of integers

**Problem:** Find longest sequence of integers \( i_1 < i_2 < \ldots < i_k \) such that,

\[ A[i_j] < A[i_{j+1}] \] for \( j = 1, \ldots, k - 1. \)

A sequence of integers is either empty or an integer followed by a sequence of integers.
Find the “best” sequence:

Either return the empty sequence, or for every possible first element \( x \):  
- recursively compute the best sequence that can follow \( x \),  
- return the best of these sequences.

This is not quite recursive...

We need to fill in the details, what is “best”, and what elements can follow another element in the sequence:

1. The elements are from the input array \( A[1 .. n] \).
2. One sequence is better than another if it has more elements.

**LIS(A[1 .. n]):**

\[
\text{max} \leftarrow 0 \\
\text{For } i = 1, \ldots, n \text{ do} \\
\quad L \leftarrow 1 + \text{LISA}\text{fter}(A[i], A[i+1 \ldots n]) \\
\quad \text{If } L > \text{max} \text{ Then} \\
\quad \quad \text{max} \leftarrow L \\
\text{Return max}
\]

**LISA\text{fter}(x, A[1 \ldots n]):**

\[
\text{If } n \leq 0 \text{ Then return 0} \\
\text{max} \leftarrow 0 \\
\text{For } i = 1 \ldots n \text{ do} \\
\quad \text{If } A[i] > x \text{ Then} \\
\quad \quad L \leftarrow 1 + \text{LISA}\text{fter}(A[i], A[i+1 \ldots n]) \\
\quad \quad \text{If } L > \text{max} \text{ Then} \\
\quad \quad \quad \text{max} \leftarrow L \\
\text{Return max}
\]

We can clean up the code a bit...

**LIS(A[1 .. n]):**

\[
A[0] \leftarrow -\infty \\
\text{Return LISA}\text{fter}(A[0 \ldots n])
\]
LISAfter( A[0...n])
If n \leq 0 Then return 0
max ← 0
For i = 1...n do
  If A[i] > A[0] Then
    L ← 1+LISAfter(A[i...n])
  If L > max Then
    max ← L
return max

Whats the running time of LIS? 
T(n)- running time of LISAfter

\[ T(n) \leq O(n) + \sum_{i=1}^{n} T(n-i) \]
\[ \leq O(n) + \sum_{i=0}^{n-1} T(i). \]

How to solve such a recurrence? 
Let us define \( S(n) = \sum_{i=0}^{n} T(i) \). Then 

\[ T(n) \leq O(n) + \sum_{i=0}^{n-1} T(i). \]
\[ S(n) - S(n-1) \leq O(n) + S(n-1) \]
\[ S(n) \leq O(n) + 2S(n-1) \]

Which implies:

\[ S(n) \leq O(n) + 2O(n-1) + 4S(n-2) \]
\[ \leq \sum_{i=0}^{k-1} 2^i O(n-i) + 2^k S(n-k) \]

by repeated opening of the recurrence. Now, set \( k = n \). We get

\[ S(n) \leq \sum_{i=0}^{n-1} 2^i O(n-i) + 2^n S(0). \]

The summation behaves like a geometric series, and is dominated by the last element, which is \( 2^{n-1}O(1) \). We also know that \( S(0) = T(0) = O(1) \). Thus, 

\[ S(n) \leq O(2^n) + O(2^n) = O(2^n). \]

Since \( T(n) \leq S(n) \), it follows that \( T(n) = O(2^n) \).
4 Designing recursive algorithms

Recursion is reduction from a problem to a simpler version of the same problem. If recursion is confusing, pretend that the simpler problem will be solved by someone else, whom we’ll call the Recursion Fairy. Recursive algorithms (and inductive proofs) closely follow recursive definitions. Here are a few recursive definitions that we can used to derive recursive algorithms:

- A natural number is either $0$ or the successor of a natural number.
- A sequence is either empty or an atom followed by a sequence.
- A sequence of length $n$ is either empty, an atom, or a sequence of length $\lfloor n/2 \rfloor$ followed by a sequence of length $\lceil n/2 \rceil$.
- A set is either the empty set or the union of a set and an atom.
- A nonempty set either is a singleton or the union of two nonempty sets.
- A binary tree is either nothing or a node pointing to two binary trees.
- A full binary tree is either just a node, or a node pointing to two full binary trees.
- A full binary tree is either just a node, or a full binary tree with two nodes glued to one of its leaves.
- A full binary tree is either a node, or a full binary tree where one leaf has been replaced by a full binary tree.
- A triangulated polygon is nothing, or a triangle glued to a triangulated polygon [not obvious]
- A triangulated polygon is nothing, or a triangle glued to two triangulated polygons [even less obvious] $x$
- A string of balanced parens is either nothing, or two strings of balanced parens, or a string of balanced parens inside a pair of parens.