LECTURE 3 (September 2nd) → Administrivia HW0 due, HW1 released

group of ≤ 3, collaboration

CONVOLUTION & BACKTRACKING

Last time we saw the Fast Fourier Transform Algorithm and how it can be used to multiply two degree-n polynomials in $O(n \log n)$ time

We also saw that the operation of polynomial multiplication can be abstracted as a convolution of two arrays which can then be computed quickly using FFT

Given vectors $A[0,\dots n]$ & $B[0,\dots n]$, the convolution $A * B[0,\dots 2n]$ is the vector

$$A * B[k] = \sum_{i+j=k} A[i] \cdot B[j]$$ ← coefficient of x^k in $A(x) \cdot B(x)$

This can be interpreted as multiplying two polynomials whose coefficients are given by A & B respectively

But from the point of view of algorithms, there are many other applications of FFT & convolutions, for instance, integer multiplication but integer and polynomial multiplication are somewhat similar tasks (except for keeping track of overflow and carry on which is very difficult)

Let's see another application of convolution that will be useful for the HW and there will be yet another application of convolution in the HW that is going to look quite different from either multiplication or the one we are going to see now

Let's suppose we are give two sets of positive integers

$$X = \{x_1, \dots x_n\}$$

$$Y = \{y_1, \dots y_n\}$$

Define the set $X+Y = \{x+y \mid x \in X \text{ and } y \in Y\}$

Each different sum appears once in the set, e.g. if

$X = \{1, 3\}$ and $Y = \{2, 4\}$, then $X+Y = \{3, 5, 7\}$

Let's see if we can compute how big is $X+Y$?

How can we compute it ?

Easy: via hashing in $O(n^2)$ time

via balanced binary search tree in $O(n^2 \log n)$ time

We are going to see an algorithm that is much faster under some circumstances by expressing this problem as convolutions

Think of sets X and Y as bit vectors

$$X[i] = \begin{cases} 1 & \text{if } i \in X \\ 0 & \text{otherwise} \end{cases} \triangleq [i \in X]$$

Let's see what $(X+Y)[k]$ is in terms of bit vectors X and Y

$$(X+Y)[k] = [k \in X+Y]$$

This is almost the convolution

$$X * Y[k] = \sum_{i+j=k} X[i] \cdot Y[j] = \begin{array}{l}\#\text{ of pairs } i \in X \\ \text{and } j \in Y \text{ that} \\ \text{sum to } k\end{array}$$

and $k \in X+Y$ if there is at least one pair that sums to k

i.e.

$$[k \in X+Y] = [X * Y[k] > 0]$$

So, to compute the size of $X+Y$

- Compute the convolution $X * Y$
- Check how many entries in $X * Y$ are bigger than zero

Running time is $O(U \log U)$ where $U = \max(X \cup Y)$

This is significantly faster than hashing or Binary search trees when U is not too large compared to n

So, when should you use convolutions ?

The picture you should have in your mind is that you are given

two arrays and you are sliding them past each other and computing sum for each shift

How far they are shifted will give one output bit

And to compute that bit we will compute sum of product of pairs of elements from each array that is given by that shift

BACKTRACKING

PRIMVSDIGNITASINTAMTENVISCIENTIANONPOTEST
ESSERESENIMSVNTPARVAEPROPEINSINGVLISLITTERIS
ATQVEINTERPVNCTIONIBUSVERBORVMOCCVPATAE

The above is a text from a speech by Cicero written in Latin

Most of the time Latin is just written like this without any spaces
Cicero refers to a computational problem in the above text called "spaces between words"

You are given a sequence of letters and you want to break it down into a sequence of words

Let's see how we can do this algorithmically — in terms of recursion

What is a sequence of words ? Recursively,

sequence of words = nothing
OR
word + sequence of words

So, one way to do this is to say either the sequence is empty of just find the first word and just let the recursion fairy deal with the rest of the sequence

So, how to find the first word ?

We will assume that there is a library that tells you what a word is

IsWord (X[1,.... k]) → returns TRUE if the sequence of letters X[1,.. k] is a word

So, how do we find the first word using the above ?

There are two conditions that we need to check

[1] The prefix we have chosen is a word → There is a library function

[2] Rest of the strings is splittable into words → We don't to think about it "Recursion"

But still we have not answered the question, how do we find the right prefix ? Just try all prefixes

```
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
```

This is backtracking otherwise known as Recursive Brute-force

What is the running time ?

How many ways are there to split a string into substring ?

After each character, we can make two choices to include it or not so, 2^n choices and this algorithm explores all choices

So, this takes $O(2^n)$ time in the worst-case

Where is the source of inefficiency in this algorithm ?

The algorithm is searching over the same space over and over

For example, suppose the algorithm is trying to break the following sequence

NOWHEREISTHISCLEARER

It is going to first break NOW + HERE + Recurse on the rest but in the top-level, it is also going to try NOWHERE + Recursion

In both cases, it recurses on the same suffix here which in the worst-case could take exponential time and this algorithm is doing exactly the same work twice. This is what makes it exponential time.

To avoid it, instead of passing the whole suffix, we are only going to pass around the index where the suffix starts.

So, consider the following function which returns TRUE if A[i...n] can be split into words

$$Splittable(i) = \begin{cases} \textsc{True} & \text{if } i > n \\ \bigvee_{j=i}^{n} \left(\textsc{IsWord}(i,j) \wedge Splittable(j+1) \right) & \text{otherwise} \end{cases}$$

```
⟨⟨Is 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
```

How many possible suffixes are there? Exactly n.

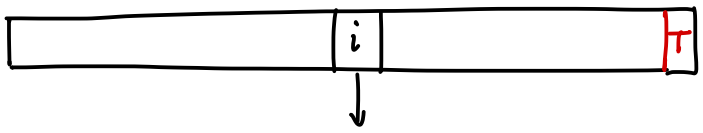
So, why are we spending exponential time? Instead the first time we see a suffix j, we are going to write it down in an array and check if we have computed this before.

This is called memoization. We can memoize this function into a data structure

INPUT : i

OUTPUT : SPLITABLE(i) OR ???

ARRAY : SPLITABLE [1, ..., n+1]



Value of SplitTable[i]

How will this table be filled ? The base case is when we are at the end at the last symbol.

In intermediate recursion steps, say while we are computing SplitTable[i] we will need to compute values of SplitTable[j] where $j > i$

So, this means that the table is going to be filled right to left automatically.

But this also means that we don't have to do recurse at all. We can just fill the table ourselves from right to left using a for loop. This is called "dynamic programming"

```
FastSplittable(A[1..n]):
  SplitTable[n+1] ← True
  for i ← n down to 1
      SplitTable[i] ← False
      for j ← i to n
          if IsWord(i, j) and SplitTable[j+1]
              SplitTable[i] ← True
  return SplitTable[1]
```

Running Time = $O(n^2)$

Fundamentally, this is based on the same recursion but we made the recursion efficient by doing it in the right order of sub-problems and storing it in the right data structure.

So, dynamic programming is not about nested for loops & table but smart recursion.

LONGEST INCREASING SUBSEQUENCE

Given a sequence of numbers, find a subsequence that is increasing and the longest

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

Let's follow the recursive idea we used previously. Consider the recursive definition:

increasing sequence = empty
OR
number + increasing sequence

This is not enough! It doesn't enforce that the sequence is increasing.

Consider a modified definition:

increasing sequence = empty
OR
number (y > x) + increasing sequence (> y)

This is my job — Recursion fairy takes care of it

So, now for our input sequence, we need to identify for every item whether its in the longest increasing subsequence. Let's jump into the middle and assume a few of the first items have already been identified.

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

Decision already made

What about this? Is it in LIS? So, 5 is not in LIS

How about 8? Not easy to tell! What do we do?

RECURSE in both cases, assuming 8 ∈ LIS and one assuming 8 ∉ LIS

That's the intuitive idea behind this algorithm.

But now when we recurse, we can't just recurse on the suffix itself. We also need to pass and remember some additional information about which case we are in

To keep track of it, what do we need to remember?

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

This does not matter

Only need to remember the largest number in the subsequence so far

Consider the recursion :

LISbigger (i,j) = Length of LIS of A[j....n] all bigger than A[i]

↳ We already saw the recursive definition of this

We need to check if A[j] is in this subsequence.

What does this function look like ?

Last # we have decided is in in LIS

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

If j^{th} number is smaller then we start looking at j+1 but have the same largest # A[i]

In this we try both, whether

A[j] ∉ LIS OR A[j] ∈ LIS

You can also write it this recursion in pseudocode as well :

```
LISBIGGER (i,j):

    if j>n : return 0

    else if A[i] ≥ A[j] : return LISBIGGER (i,j+1)

    else : skip ← LISBIGGER (i,j+1)
           take ← LISBIGGER (j, j+1) + 1
           return max {skip, take}
```

This gives an algorithm to find the LIS bigger than some value x.
How do we solve the original problem?

One way: add an artificial value for x, say $-\infty$

Another way: in a for loop, try all possible initial value for x

But wait, this might take exponential time: we are only reducing the problem size by one but potentially recursing on each sub-problem twice.

How do we memoize this function to get an efficient algorithm?

↳ NEXT TIME