Basic Algorithms Lec-5

Abhinav Tamaskar
12 June, 2019
CIMS, NYU
Diffie-Hellman key exchange
Alice and Bob are sending messages over an insecure channel, where Eve is eavesdropping on their communication. How do we encrypt messages so that no one can decipher them?
To encrypt messages using any cryptographic scheme, we first need to have a common encryption key.
How do we manage to share a key, without letting Eve see it?
To encrypt messages using any cryptographic scheme, we first need to have a common encryption key.
How do we manage to share a key, without letting Eve see it?
We use modular arithmetic!
Definition
A primitive root modulo $n$ is a number $g$ such that

$$\{g, g^2, g^3, \ldots, g^{n-1}\} = \{1, 2, \ldots, n-1\} = \mathbb{Z}_n^*$$

E.g. for $n = 7$, 3 is a primitive root as

$$\{3, 3^2 = 2, 3^3 = 6, 3^4 = 4, 3^5 = 5, 3^6 = 1\}$$

Another primitive root for 7 is 5.
Theorem
Every prime has a primitive root.

E.g. some of the first few primes and their roots are:

<table>
<thead>
<tr>
<th>Prime</th>
<th>Roots</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2, 3</td>
</tr>
<tr>
<td>7</td>
<td>3, 5</td>
</tr>
<tr>
<td>11</td>
<td>2, 6, 7, 8</td>
</tr>
<tr>
<td>13</td>
<td>2, 6, 7, 11</td>
</tr>
</tbody>
</table>
The Diffie-Hellman protocol works as follows:

1. Alice and Bob publicly choose an appropriate prime $p$ and a primitive root $g$.

2. Alice chooses a secret number $a$ and Bob chooses a secret number $b$.

3. Alice sends $g^a$ to Bob and Bob sends $g^b$ to Alice.

4. Alice and Bob both calculate $g^{ab} := k$, which is their secret key.
This relies on the discrete log assumption.

Definition
Discrete log assumption. Given a prime $p$, a primitive root $g$ and an element $x \in \mathbb{Z}_p$, calculating $a$ such that $g^a = x$ is hard.

Here hard means that there is no known algorithm which can run in poly-time, i.e. in time polynomial in the input size, which is $O(\log n)$, remember bit-level complexity.
If there is any algorithm which runs in $O(\log n^d)$ for any $d$, almost all modern crypto systems would break.
Dynamic Programming
DP: a **structured** way of doing calculations by reusing recursive calls.

Original term coined by Richard Bellman in 1950, to confuse politicians.
For more etymology, do some bedtime reading on stack-exchange
Problem
Change making problem: Given coins of $n$ values, $c_1, c_2, \ldots, c_n$ and a final amount $P$ find the minimum number of coins needed to get $P$.
E.g. in the US system

\{1, 2, 5, 10, 20, 50, 100\}

\[70 \rightarrow 50 + 20\]
\[80 \rightarrow 50 + 20 + 10\]
\[90 \rightarrow 50 + 20 + 20\]
Naive strategy: Pick the largest denomination possible and continue recursively.

90 → pick 50
   40 → pick 20
      20 → pick 20

This seems to work!
Counter-example: \(\{1, 5, 8\}, P = 10\)

Using the naive strategy:

\[
10 \rightarrow 8 + 1 + 1
\]

Better solution

\[
10 \rightarrow 5 + 5
\]

So even when using recursion cannot discount the fact that we might be using smaller denominations, making the naive strategy incorrect.
Counter-example:

\( \{1, 5, 8\}, P = 10 \)

Using the naive strategy:

\[ 10 \rightarrow 8 + 1 + 1 \]

Better solution

\[ 10 \rightarrow 5 + 5 \]

So even when using recursion cannot discount the fact that we might be using smaller denominations, making the naive strategy incorrect. In fact, the currency systems are designed so that the naive strategy works.
Better solution: Use recursion, but try all possible denominations at each step.
Algorithm 1 Recursive change making

```plaintext
function change_r(Int P, Int coin[])
    if P in coins then
        return 1
    elseif P smaller than all coins then
        return ∞
    else
        return 1 + \min_{i; P \geq coin[i]} change_r(P-coin[i])
    end if
end function
```
Time complexity:
For that we need to look at the total amount of recursive calls made.

The first leaf will occur at height \( h = \left\lfloor \frac{p}{\text{coin}[n]} \right\rfloor \), which gives \( O(n^h) \) number of recursive calls.
This is way too much....
Can we do better?
Look at the repeated calls.

Which means that we are repeating too many calls. We can do better by storing them in a table.
Store a table: \( DP[1 \ldots P] \)

\[
DP[x] := \text{minimum number of coins needed to get } x
\]

We can now look at how to calculate \( DP[x] \) based on smaller values in the table, based on the recursive approach.

\[
DP[x] = \min_{i; x \geq \text{coin}[i]} 1 + DP[x - \text{coin}[i]]
\]

E.g. for \( P = 10 \), coins = \{1, 5, 8\}
Time complexity:
Time taken to fill on block of the DP table: $O(n)$
Size of the table: $O(P)$
Total time: $O(nP)$. 
Time complexity:
Time taken to fill on block of the DP table: $O(n)$
Size of the table: $O(P)$
Total time: $O(nP)$.
These is called pseudo-polynomial in the input size, as the input $P$ is of size $O(\log P)$. 
Problem

0-1 Knapsack: Given $n$ items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of a maximum capacity weight $W$, find the maximum value of items you can fit in the knapsack. E.g. for a knapsack of max capacity 22, we have

weights: 2 5 7 10 11 13

values: 3 10 8 22 23 28

which gives us the two items $(10, 22)$ and $(11, 23)$ to give a max value of 45.
Already we see that we cannot choose the maximum valued item. We can again try to solve the problem recursively by choosing an item and trying to solve the smaller problem.
Algorithm 2 0-1 Knapsack

function knapsack_r(Int weight[], Int value[], Int W)
    \(v_1 \leftarrow 0\)
    if \(weight[1] \leq W\) then
        \(v_1 \leftarrow value[1] + \text{knapsack}_r(weight[2:], value[2:], W-weight[1])\)
    end if
    \(v_2 \leftarrow \text{knapsack}_r(weight[2:], value[2:], W)\)
    return max\((v_1, v_2)\)
end function
Similar to the previous analysis, we see that time complexity of the recursive algorithm is exponential.
What table do we make?
Identifying subproblems is an important skill when solving DP problems.
Inputs to the recursive function are of the form

\[ [w_1, \ldots, w_k], [v_1, \ldots, v_k], C \]

where \( k : 0 \rightarrow n, C : 0 \rightarrow W \)

Make a 2-D table: \( DP[0 \ldots n][0 \ldots W] \)

\[ DP[k][C] := \text{maximum value possible using the } \]
\[ \text{first } k \text{ items for a knapsack of capacity } C \]

And we get the recursion as

\[ DP[k][C] = \max(DP[k - 1][C], \quad DP[k - 1][C - \text{weight}[k]] + \text{value}[k]) \]

And the time complexity for this solution is \( O(nW) \)
Problem
Gene sequence alignment: Given two gene sequences, $s_1 \ldots s_n, t_1 \ldots t_m$ find an optimal alignment for the sequences.
What is an alignment?

Try to line-up the sequences so that most characters match up. The score is a +2 for every match and a 1 for every mismatch.

E.g. (a) for ACGTAAT and GACTTA an alignment would be of the form _ACGTAAT GACTTA__ with a score of 4.

(b) for the sequences AGAAGGT and CGAATTAAG we can have AG_A__AGGT CGAATTAAG_ with a score of 2.
What is an alignment?
Try to line-up the sequences so that most characters match up. The score is a $+2$ for every match and a $-1$ for every mismatch.

E.g. (a) for ACGTAAT and GACTTA an alignment would be of the form

```
_ ACGTAAT
  GACTTA__
```

with a score of 4.
(b) for the sequences AGAAGGT and CGAATTAAG we can have

```
AG_A__AGGT
  CGAATTAAG_
```

with a score of 2
It is possible to have more than one optimal alignment. E.g. for the first pair of sequences ACGTAAT and GACTTA we could also have

_AC_GTAAT
GAC_TTA_
In real world biology, a more complicated scoring matrix is used,

\[ \delta : (\Sigma \cup \{ _\} )^2 \to \mathbb{R} \]

Here the score to the final sequences \( s'_1 \ldots s'_k, t'_1 \ldots t'_k \) is given by

\[ \sum_{i=1}^{k} \delta(s_i, t_i) \]
How do we find the alignment with the largest score?
How do we find the alignment with the largest score?
Thinking recursively: Let us first align the last two elements $s_n$, $t_m$.
There are three possibilities for $s_n$, $t_m$:

1. align them to each other
2. align $s_n$ to a _
3. align $t_m$ to a _

And then we can continue with the smaller sequences
Store a 2-D table: $DP[1 \ldots n][1 \ldots m]$

$DP[i][j] := \text{maximum alignment score of } s_1 \ldots s_i, t_1 \ldots t_j$

and we can calculate it recursively as

$$DP[i][j] = \max \begin{cases} 
DP[i - 1][j - 1] + \delta(s_i, t_j) \\
DP[i - 1][j] + \delta(s_i, \_ ) \\
DP[i][j - 1] + \delta(\_, t_j) 
\end{cases}$$
How do we actually find the alignment instead of just the score?
How do we actually find the alignment instead of just the score? Store a path matrix, $Path[1 \ldots n][1 \ldots m]$. 
How do we actually find the alignment instead of just the score? Store a path matrix, $Path[1\ldots n][1\ldots m]$. In the path matrix, we store which of the three cases gave the maximum, which allows us to reconstruct the “path” to the final block $n \times m$. 
E.g. for the sequences ACGAC and AGC we get the two matrices

**DP:**

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>-1</th>
<th>-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

**Path:**

<table>
<thead>
<tr>
<th></th>
<th>_</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Second programming assignment:
Implement gene sequence alignment and output an optimal alignment.