CS 341: ALGORITHMS

Lecture 9: greedy algorithms III / dynamic programming I

Readings: see website

Trevor Brown

https://student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca
RETURNING TO RATIONAL KNAPSACK

- What if we **cannot assume distinctness** for profit/weight ratios in the proof?
- There is no longer a unique optimal solution
- So **cannot** prove optimal Y must be **identical** to greedy X
- Swapping might not improve the solution
OPTIMALITY PROOF WITHOUT DISTINCTNESS

- There may be many optimal solutions
- **Key idea:** Let $Y$ be an optimal solution that **matches** $X$ **on a maximal number of indices**
- **Observe:** if $X$ is really optimal, then $Y = X$
  - Suppose not for contra
- We will modify $Y$ to make it match $X$ **on one more index** (a contradiction!)
- As before, let $j$ be the first index where $X$ and $Y$ differ
Greedy solution $X$

Optimal solution $Y$

$j =$ first index where the solutions differ

$y_j \neq x_j$
Greedy solution $X$

Optimal solution $Y$

Fraction of item in knapsack

$0 \leq x_1 \leq x_2 \leq \cdots \leq x_j \leq \cdots \leq x_n \leq 1$

$0 \leq y_1 \leq y_2 \leq \cdots \leq y_{j-1} \leq y_j \leq \cdots \leq y_n \leq 1$

Must have $y_j < x_j$
Must exist \( k > j \) such that \( y_k > x_k \) because weight of \( X \) and \( Y \) must be the same.

**Remove** some weight \( \delta \) of item \( k \) and **replace** it with some of item \( j \)

With the goal of making the solutions equal on index \( k \) or index \( j \)

**Optimal solution \( Y \)**

**Fraction** we should add to \( j \) to make solutions equal on index \( j \): \( x_j - y_j \)

**Weight** to add: \( w_j(x_j - y_j) \)

**Greedy solution \( X \)**

**Fraction** we should remove from \( k \) to make solutions equal on index \( k \): \( y_k - x_k \)

**Weight** to remove: \( w_k(y_k - x_k) \)

Let \( \delta = \min\{w_j(x_j - y_j), w_k(y_k - x_k)\} \)

Observe \( \delta > 0 \)
In the case pictured here, we have $\delta = w_k(y_k - x_k)$, so we end up with $y'_k = x_k$. 

### Optimal solution $\mathbf{Y}$

- $\delta = \min\{w_j(x_j - y_j), w_k(y_k - x_k)\}$
- $w_j(x_j - y_j)$
- $w_k(y_k - x_k)$

### Modified optimal solution $\mathbf{Y}'$

- $y'_j = y_j + \frac{\delta}{w_j}$
- $y'_k = y_k - \frac{\delta}{w_k}$
To show $Y'$ is feasible, we show $\text{weight}(Y') \leq M$ and $y'_k \geq 0, y'_j \leq 1$

**Weight**

We move $\delta$ weight from item $k$ to item $j$

This does not change the total weight!

So $\text{weight}(Y') = \text{weight}(Y) = M$
FEASIBILITY OF $Y'$

- Showing $y'_k \geq 0$
  - By definition, $y'_k = y_k - \frac{\delta}{w_k} \geq 0$ iff $\delta \leq y_k w_k$
  - But $\delta$ is the minimum of $w_k(y_k - x_k) \leq w_k y_k$ and another expr.
  - So $\delta \leq y_k w_k$

- Showing $y'_j \leq 1$
  - $y'_j = y_j + \frac{\delta}{w_j} \leq 1$ iff $\frac{\delta}{w_j} \leq 1 - y_j$ iff $\delta \leq w_j(1 - y_j)$ (rearranging)
  - $\delta \leq w_j(x_j - y_j)$ (definition of $\delta$)
  - $\delta \leq w_j(x_j - y_j) \leq w_j(1 - y_j)$ (by feasibility of $X$)
PROFIT OF $Y'$

- $profit(Y') = profit(Y) + \frac{\delta}{w_j} p_j - \frac{\delta}{w_k} p_k = profit(Y) + \delta \left( \frac{p_j}{w_j} - \frac{p_k}{w_k} \right)$

- Since $j$ is before $k$, and we consider items with more profit per unit weight first, we have $\frac{p_j}{w_j} \geq \frac{p_k}{w_k}$.

- Since $\delta > 0$ and $\frac{p_j}{w_j} \geq \frac{p_k}{w_k}$, we have $\delta \left( \frac{p_j}{w_j} - \frac{p_k}{w_k} \right) \geq 0$

- Since $Y$ is optimal, this cannot be positive

- So $Y'$ is a new optimal solution that matches $X$ on one more index than $Y$

- Contradiction: $Y$ matched $X$ on a maximal number of indices!
DYNAMIC PROGRAMMING
Sort of like divide and conquer... but sometimes better
Richard Bellman, the inventor of dynamic programming in 1950, related the following in his autobiography:

“What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, ‘programming.’ I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, lets kill two birds with one stone. Lets take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is its impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. Its impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.”

“Bottom-up recursion” might also a reasonable name, as we’ll see...
COMPUTING FIBONACCI NUMBERS \textbf{INEFFICIENTLY}
A TOY EXAMPLE TO COMPARE D\&C TO DYNAMIC PROGRAMMING

1. \textbf{BadFib}(n)
2. \hspace{1em} \textbf{if} \ n = \text{0} \ \textbf{or} \ n = \text{1} \ \textbf{then return} \ n
3. \hspace{1em} \textbf{return} \ \textbf{BadFib}(n-1) + \ \textbf{BadFib}(n-2)

\textbf{FIBONACCI PIGEONS}
**RUNTIME**

- **In unit cost model**
  - *(UNREALISTIC!)*
  - \( T(n) = T(n-1) + T(n-2) + O(1) \)
  - \( T(n) \geq 2T(n-2) + O(1) \)
  - \( T(n) \leq 2T(n-1) + O(1) \)
  - \( n/2 \) levels of recursion for the first expression
  - \( n \) levels for the second expression
  - Work doubles at each level
  - \( T(n) \) is certainly in \( \Omega(2^{n/2}) \) and \( O(2^n) \)

---

**Code Snippet**

```
1 BadFib(n)
2   if n == 0 or n == 1 then return n
3   return BadFib(n-1) + BadFib(n-2)
```

This \( O(1) \) would change in the bit complexity model.

This is exponential in \( n \).
Is it exponential in the input size?
WHAT IS THE INPUT SIZE?

- Input: n
- Bits to store n? 
  - $\lceil \log n \rceil$
- For simplicity say input size is $S = \log n$
- So $2^S = 2^{\log n} = n$
- So $\frac{n}{2} = 2^{S-1}$
- Therefore, $T(n) \in \Omega(2^{n/2}) \Rightarrow T(n) \in \Omega(2^{2^{S-1}})$
- Recall $T(n) \in O(2^n) = O(2^{2S})$

In most of our analyses before this point, $n$ has coincidentally been the size of the input.
WHY IS THIS **SO** SLOW?

- Subproblems have **LOTS** of overlap!
- Every subtree on the right appears on the left
- ... recursively ...
- Each subtree is computed **exponentially** often in its depth

This overlap suggests dynamic programming may be able to help!
Designing Dynamic Programming Algorithms for Optimization Problems

(Optimal) Recursive Structure
Examine the structure of an optimal solution to a problem instance $I$, and determine if an optimal solution for $I$ can be expressed in terms of optimal solutions to certain subproblems of $I$.

Define Subproblems
Define a set of subproblems $S(I)$ of the instance $I$, the solution of which enables the optimal solution of $I$ to be computed. $I$ will be the last or largest instance in the set $S(I)$.

Or, if it’s not an optimization problem, simply “determine if a solution for $I$ can be expressed in terms of solutions to certain subproblems of $I”.

“... which enables the solution of $I$ to be computed”
Designing Dynamic Programming Algorithms (cont.)

Recurrence Relation
Derive a recurrence relation on the optimal solutions to the instances in $S(I)$. This recurrence relation should be completely specified in terms of optimal solutions to (smaller) instances in $S(I)$ and/or base cases.

Compute Optimal Solutions
Compute the optimal solutions to all the instances in $S(I)$. Compute these solutions using the recurrence relation in a bottom-up fashion, filling in a table of values containing these optimal solutions. Whenever a particular table entry is filled in using the recurrence relation, the optimal solutions of relevant subproblems can be looked up in the table (they have been computed already). The final table entry is the solution to $I$. 
SOLVING FIB USING DYNAMIC PROGRAMMING

- (Optimal) Recursive Structure
  - Solution to $n$-th Fibonacci number $f(n)$ can be expressed as the addition of smaller Fibonacci numbers
  - No notion of optimality for this particular problem

- Define Subproblems
  - The set subproblems that will be combined to obtain $Fib(n)$ is $\{Fib(n-1), Fib(n-2)\}$
  - $S(I) = \{Fib(0), Fib(1), ..., Fib(n)\}$

- Recurrence Relation
  $$\begin{aligned}
  f(n) &= \begin{cases}
  f(n-1) + f(n-2) & : i \geq 2 \\
  1 & : i = 1 \\
  0 & : i = 0
  \end{cases}
  \end{aligned}$$

- Computing (Optimal) Solutions
  - Create table $f[1..n]$ and compute its entries “bottom-up”
FILLING THE TABLE “BOTTOM-UP”

- Key idea:
  - When computing a table entry
  - Must have **already computed** the **entries** it depends on!

- Dependencies
  - Extract directly from recurrence
  - Entry n depends on n-1 and n-2

- **Computing entries in order 1..n** guarantees n-1 and n-2 are already computed when we compute n
DP SOLUTION

- **Space saving** optimization:
  - We never look at \( f[i-3] \) or earlier
  - Can make do with a few variables instead of a table

```java
FibDP(n)
    f = new array of size n
    f[0] = 0
    f[1] = 1
    for i = 2..n
        f[i] = f[i-1] + f[i-2]
    return f[n]
```

**FibDP(n)**

```
fi2 = 0
fi1 = 1
for i = 2..n
    temp = fi
    fi = fi1 + fi2
    fi2 = fi1
    fi1 = temp
return fi
```

- Represents \( f[i-2] \)
- Represents \( f[i-1] \)
- Save \( f[i] \) before overwriting it (so its value can be stored in \( f[i-1] \) later)
- In next iteration, \( f[i-2] \) will be the current \( f[i-1] \)
- And \( f[i-1] \) will be the current \( f[i] \)
- This is still considered to be dynamic programming...
  We’ve just optimized out the table.
CORRECTNESS

- **Step 1**
  - Order 0..n means i-1 and i-2 are already computed when we compute i
  - Prove that when computing a table entry, dependent entries are **already computed**

- **Step 2** (similar to D&C)
  - Suppose subproblems are solved correctly (optimally)
  - Prove these (optimal) subsolutions are combined into a(n optimal) solution
  - Suppose f[i-1] and f[i-2] are the (i-1)th and (i-2)th Fib #s
  - Then f[i] = the n-th Fib #

```
1 FibDP(n)
2     f = new array of size n
3     f[0] = 0
4     f[1] = 1
5     for i = 2..n
6         f[i] = f[i-1] + f[i-2]
7     return f[n]
```
Unit cost model is **not realistic** for this problem, because Fibonacci numbers grow quickly:

- $F[0]=0$
- $F[10]=55$
- $F[100]=354224848179261915075$
- $F[300]=222232244629420445529739893461909967206$
  
  $666939096499764990979600$
- $F[1000]=\text{more than 200 digits}$
How quickly does $f_n$ grow? Let $\phi = (1 + \sqrt{5})/2$; then

$$f_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}} = \left[ \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right].$$

Therefore $f_n \in \Theta(\phi^n)$ and hence we also have $f_{n+1} \in \Theta(\phi^n)$.

The value $\phi \approx 1.6$ is the golden ratio.

- Value of $f[n]$ is exponential in $n$
- So **number of digits** of $f[n]$ is **linear in n**
- Big numbers suggest using **bit-complexity model**
RUNNING TIME

BIT-COMPLEXITY MODEL

- \( f[i-2], f[i-1] \) have \( \Theta(i) \) digits
- So \( f[i-1]+f[i-2] \) takes \( \Theta(i) \) time
- \( T(n) \in \sum_{i=1}^{n} \Theta(i) = \Theta(n^2) \)

Is this \textbf{quadratic runtime}?

- NO! This is “quadratic in n”
- When we say “quadratic runtime” we mean “quadratic in the input size.”
- What’s the input size \( S \)?
  - The input is the number \( n \), so \( S = \log n \) bits

\[
\begin{align*}
2^S &= 2^{\log n} = n \\
\text{So } \Theta(n^2) &= \Theta \left( (2^S)^2 \right)
\end{align*}
\]

This algorithm is still \textbf{exponential in the input size}!

... but it’s \textbf{not} doubly exponential in \( S \).

\textbf{Exponential improvement!}
TIPS FOR ANALYSIS OF DP ALGORITHMS

• Think carefully about which model of computation (unit cost / bit complexity) is appropriate
  ◦ If you can’t decide which is appropriate, you can try both and see if it changes the answer

• Think carefully about the input size S
  ◦ Try to express runtime in terms of S
  ◦ If that’s too hard, try to find an elegant/natural expression (see future lectures)
  ◦ An algorithm is “linear time” only if it’s “linear in S”
OTHER MISCELLANEOUS TIPS

• Building a table of results bottom-up is what makes an algorithm DP

• There is a similar concept called **memoization**
  - But, for the purposes of this course, we want to see bottom-up table filling!

• Base cases are **critical**
  - They often completely determine the answer
  - Try setting $f[1]=0$ in FibDP…

```cpp
FibDP(n) {  
  f = new array of size n  
  f[0] = 0  
  f[1] = 1  
  for $i = 2..n$  
    $f[i] = f[i-1] + f[i-2]$  
  return $f[n]$  
}
```
ROD CUTTING
A “REAL” DYNAMIC PROGRAMMING EXAMPLE

- Input:
  - \( n \): length of rod
  - \( p_1, \ldots, p_n \): \( p_i = \text{price of a rod of length } i \)

- Output:
  - Max \textit{income} possible by cutting the rod of length \( n \) into any number of \textit{integer} pieces (maybe \textit{no} cuts)

<table>
<thead>
<tr>
<th>length ( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>price ( p_i )</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

\( n = 4 \)

All ways of cutting a rod of length 4

Example output: 10
DYNAMIC PROGRAMMING APPROACH

- High level idea *(can just think recursively to start)*
  - Given a rod of length \( n \)
  - Either make no cuts, or make a cut and **recurse** on the remaining parts

- **Where** should we cut?
DYNAMIC PROGRAMMING APPROACH

- Try **all ways** of making that cut
  - i.e., try a cut at positions 1, 2, ..., \( n - 1 \)
  - In each case, recurse on two rods \([0, i]\) and \([i, n]\)
- Take the max income over **all possibilities** (each \( i \) / no cut)

\[
\begin{align*}
i = 1 & \quad \vdots \\
i = 2 & \quad \vdots \\
i = 3 & \quad \vdots \\
i = n - 1 & \quad \vdots
\end{align*}
\]

Optimal substructure:
- Max income from two rods w/sizes \( i \) and \( n - i \)
- ... is max income we can get from the rod size \( i \)
- + max income we can get from the rod size \( n - i \)
RECURRENCE RELATION

- Define $M(k) = \text{maximum income for rod of length } k$
- If we do **not** cut the rod, max income is $p_k$
- If we **do** cut a rod at $i$

- max income is $M(i) + M(k - i)$
- Want to maximize this **over all** $i$
  - $\max_i\{M(i) + M(k - i)\}$ (for $0 < i < k$)
  - $M(k) = \max\{p_k, \max_{1 \leq i \leq k-1}\{M(i) + M(k - i)\}\}$
**Computing Solutions Bottom-Up**

- **Recurrence:** $M(k) = \max\{p_k, \max_{1 \leq i \leq k-1} \{M(i) + M(k - i)\}\}

- **Compute table of solutions:** $M[1..n]$

  $\begin{array}{|c|c|c|c|c|}
  \hline
  & 1 & \cdots & k & n \\
  \hline
  M & \text{fill} & \text{fill} & \text{fill} & \text{fill} \\
  \hline
  \end{array}$

- **Dependencies:** entry $k$ depends on
  - $M[i] \rightarrow M[1..(k - 1)]$
  - $M[k - i] \rightarrow M[1..(k - 1)]$

- All of these dependencies are $< k$

- So we can fill in the table entries in order $1..n$
Recurrence: \( M(k) = \max\{p_k, \max_{1 \leq i \leq k-1} \{M(i) + M(k-i)\}\} \)

Recall \( M(k) = \) maximum income for rod of length \( k \)

```python
def RodCutting(n, p[1..n]):
    M = new array[1..n]

    // compute each entry M[k]
    for k = 1..n
        M[k] = p[k] // current best = no cuts

    // try each cut in 1..(k-1)
    for i = 1..(k-1)
        M[k] = max(M[k], M[i] + M[k-i])

    return M[n]
```

Time complexity? \( \Theta(n^2) \)
Is this a quadratic time algorithm?
Unit cost model is appropriate here

Each element of \( p \) takes \textbf{one word}

\((\Theta(1) \text{ bits each})\)

So \( \Theta(n) \) words

\textbf{Input size} \( S \in \Theta(n) \)

```java
RodCutting(n, p[1..n])
M = new array[1..n]

// compute each entry M[k]
for k = 1..n
    M[k] = p[k] // current best = no cuts

// try each cut in 1..(k-1)
for i = 1..(k-1)
    M[k] = max(M[k], M[i] + M[k-i])

return M[n]
```

So with runtime \( \Theta(n^2) = \Theta(S^2) \), this \textbf{is} a quadratic time algorithm
NEXT UP...

- DP 0-1 Knapsack and Coin Changing (for all currencies)
- Tables will feature multiple dimensions
  - (Not just a 1D array)
  - Bottom-up filling orders become non-trivial
- We often want to solve optimization problems
  - Arguing that an optimal solution is built from optimal sub-solutions becomes more significant
- Input size calculations become more complex, and runtimes often include multiple variables