FINISHING UP GREEDY

PROBLEM: INTERVAL COLOURING

Instance: A set $A = (A_1, \ldots, A_n)$ of intervals, for $1 \leq i \leq n$, $A_i = ([u_i, v_i])$, where $u_i$ is the start time of interval $A_i$ and $v_i$ is the finish time of $A_i$.

Feasible solution: A $c$-colouring is a mapping $col : A \rightarrow \{1, \ldots, c\}$ that assigns each interval $i$ a colour such that two intervals receiving the same colour are always disjoint.

Find: A $c$-colouring of $A$ with the minimum number of colours.

MORE EXAMPLES

Greedy Strategies for Interval Colouring

As usual, we consider the intervals one at a time. At a given point in time, suppose we have coloured the first $i < n$ intervals using $d$ colours.

We will colour the $(i + 1)$th interval with any permissible colour. If it cannot be coloured using any of the existing $d$ colours, then we introduce a new colour and $d$ is increased by 1.

Question: In what order should we consider the intervals?
Consider intervals in the order they are given in the input: $A_1, A_2, \ldots, A_{10}$.
EXAMPLE: ORDER MATTERS!

- Example diagrams showing the effect of order in scheduling tasks.
- Diagrams illustrate different scenarios where the order of tasks affects the outcome.

Used 4 colours

Pre-sort intervals by increasing start time

Can we do better?
EXAMPLE: ORDER MATTERS!
Pre-sort intervals by increasing start time!
EXAMPLE: ORDER MATTERS!

Used 3 colours
Turns out to be optimal...

ORDER MATTERS!

EXAMPLE: ORDER MATTERS!

ORDER MATTERS!

ORDER MATTERS!

ORDER MATTERS!

ORDER MATTERS!

ORDER MATTERS!

ORDER MATTERS!
Correctness of the Algorithm

The correctness of this greedy algorithm can be proven inductively as well as by a “slick” method—we give the “slick” proof.

Let $D$ denote the number of colours used by the algorithm.
We prove $\mathcal{F}_D$ overlaps $D-1$ other intervals at a single point in time.

Let $\mathcal{F}_D$ be the first interval that has colour $D$.

We prove $\mathcal{F}_D$ overlaps every interval $\mathcal{L}_c$ for all $c < D$.

Let $\mathcal{L}_c$ be the last interval that has colour $c$ and starts before $\mathcal{F}_D$.

Note $\mathcal{L}_c$ must exist (otherwise greedy would just use colour 1 for $\mathcal{F}_D$).

And $\mathcal{L}_c \cup \mathcal{F}_D$ must be after $\mathcal{F}_D$ starts (same reason).

So, $\mathcal{F}_D$ overlaps $D-1$ intervals!

Moreover, every interval in $\{\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{D-1}\}$ contains the starting time of $\mathcal{F}_D$. So, we must use $D$ colours!

TIME COMPLEXITY?

Total $O(\log n)$ iterations if only a constant number of colours are needed (or even $\log n$ colours!).

Could be $O(n^2)$ if $n$ colours are needed.

Most accurate complexity statement is $O(n \log n + nd)$ where $D$ is # colours used.

To improve the algorithm, we could use a priority queue to keep track of the earliest finish time of each colour.

So, we only need to look at the minimum element at all times in the algorithm.

Current greedy algorithm:

• For each interval $\mathcal{A}_i$, compare its start time $s_i$ with the finish times $f_c[i]$ of all colours introduced so far.

• Why? Looking for some finish time that is earlier than $s_i$.

• We are doing linear search... Can we do better?

• Use a priority queue to keep track of the earliest finish time of each colour.

```c
GreedyIntervalColouring(s[i], f[i])
{
    colour[1] = f[1];
    for i = 2 to n
        if not revised
            revised = false
            for c = 2 to D
                if (finish[c] == nil)
                    then colour[c] = f[c];
                    revised = true;
            if not revised
                colour[c] = d
            finish[d] = f[i];
    return d;
}
```

Most inefficiencies exist in this algorithm. Could we make it faster with clever data structure usage?

EXAMPLE: HEAP-BASED ALGORITHM

Initial state

Heap

Iteration 1

Check heap minimum

Empty, so a new colour is needed.
EXAMPLE: HEAP-BASED ALGORITHM

Iteration 1
- Check heap minimum
- Empty, so a new colour is needed

Iteration 2
- Check heap minimum
- Min element: NULL
- Check if finish time 3 is before $s_2$

Iteration 3
- Check heap minimum
- Min element: NULL
- Check if finish time 3 is before $s_3$

Iteration 4
- Yes, reuse colour, deleteMin and insert new finish time into heap!
Check heap minimum

Check if finish time 3 is before $s_4$

Iteration $i = 4$

Finish at time 7

Finish at time 5

Finish at time 5

Finish at time 9

And so on, and so forth...

$O(\log S)$ where $S = \text{size(priority queue)}$

$O(1)$

$O(\log D)$

Total $O(n \log n) + O(n \log D)$

Since $n \geq D$, $O(n \log D)$

$O(1)$
We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word "research"… He would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical.

I felt I had to do something to shield Wilson… from the fact that I was really doing mathematics… What title, what name, could I choose? In the first place it was intended as planning in decision making, in thinking. But planning, is not a good word for various reasons. I decided mathematics true on words "programming." I wanted to get across the idea that this was "dynamic," this was multistage, this was time-varying. I thought, let's kill two birds with one stone.

"Bottom-up recursion" might also a reasonable name, as we'll see…

I thought dynamic programming was a good name. It was something not even a Congressman could object to.


**RUNTIME**

- In unit cost model
  - (UNREALISTIC)!
  - $T(n) = T(n-1) + T(n-2) + O(1)$
  - $T(n) \geq 2T(n-1) + O(1)$
  - $T(n) \leq 2T(n-1) + O(1)$
- $n/2$ 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 $O(2^n)$

**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!

---

**Computing Fibonacci Numbers INEFFECTIVELY**

**A TOY EXAMPLE TO COMPARE BRUTE-TO DYNAMIC PROGRAMMING**

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

**Fibonacci Pigeons**

---

**Designing Dynamic Programming Algorithms for Optimization Problems**

1. **(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 $I$.
2. **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 $S(I)$.
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 of subproblems that will be combined to obtain $\text{Fib}(n)$ is $\{\text{Fib}(0), \text{Fib}(1), ..., \text{Fib}(n)\}$
  • Recurrence Relation $f(n) = f(n-1) + f(n-2)$
  • Computing (Optimal) Solutions
    • Create table $f[1..n]$ and compute its entries “bottom-up”

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

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 $f(n)$ depends on $f(n-1)$ and $f(n-2)$
  • Computing entries in order $1..n$
    • guarantees $f(n-1)$ and $f(n-2)$ are already computed when we compute $f(n)$

CORRECTNESS

• Step 1 (similar to D&C)
  • Prove that when computing a table entry, dependent entries are already computed
  • Suppose $f[i-1]$ and $f[i-2]$ are the $(i-1)$th and $(i-2)$th Fib #s
  • Then prove $f[i] = \text{n-th Fib #}$

• Step 2 (similar to D&C)
  • Suppose subproblems are solved correctly (optimally)
  • Prove these (optimal) subsolutions are combined into an optimal solution

MODEL OF COMPUTATION FOR RUNTIME

• Unit cost model is not very realistic for this problem, because Fibonacci numbers grow quickly
  • $F[10] = 55$
  • $F[100] = 354224848179261915075$
  • $F[300] = 22223244625404655373989366198967706666693926499745497979400$
  • Value of $F[n]$ is exponential in $n$: $f(n) \in \Theta(\phi^n)$ where $\phi \approx 1.618$
  • $\phi^n$ needs $\log(\phi^n)$ bits to store it
  • So $F[n]$ needs $\Theta(n)$ bits to store it

But let’s use unit cost anyway for simplicity

RUNNING TIME (UNIT COST)

• $T(n) \in \Theta(n)$
A BRIEF ASIDE

- Is this linear runtime?
  - NO! This is “a linear function of n”
  - When we say “linear runtime” we mean “a linear function of the input size”
  - What is the input size $S$?
    - The input is the number $n$.
    - How many bits does it take to store $n$? $\Theta(\log n)$.
    - So $S = \log n$ bits

**Express $T(n)$ as a function of the input size $S$ (in bits)**

$$T(n) \in \Theta(n^2)$$

This algorithm is exponential in the input size!

ROD CUTTING

A “REAL” DYNAMIC PROGRAMMING EXAMPLE

- **Input:**
  - $n$: length of rod
  - $p_1, \ldots, p_n$: price of a rod of length $i$
- **Output:**
  - Max income possible by cutting the rod of length $n$ into any number of integer pieces (maybe no cuts)

### Example output: 10

<table>
<thead>
<tr>
<th>Length</th>
<th>$i = 1$</th>
<th>$i = 2$</th>
<th>$i = 3$</th>
<th>$i = 4$</th>
<th>$i = 5$</th>
<th>$i = 6$</th>
<th>$i = 7$</th>
<th>$i = 8$</th>
<th>$i = 9$</th>
<th>$i = 10$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

**Critical step! Must define what $M(k)$ means, semantically!**

### 74

**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
    - Income $p_k$
    - Income(left) + Income(right)
  - Where should we cut?

### 75

**Rod Cutting: A "Real" Dynamic Programming Example**

- Input:
  - $n$: length of rod
  - $p_1, \ldots, p_n$: price of a rod of length $i$
- Output:
  - Max income possible by cutting the rod of length $n$ into any number of integer pieces (maybe no cuts)

### 76

**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)

### 77

**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 $0 < i < k$)
      - $M(k) = \max(p_k, \max_i(M(i) + M(k - i)))$
**Computing Solutions Bottom-Up**

- **Recurrence:** \( M(k) = \max \{ p_i \cdot \max_{1 \leq i < k} (M(i) + M(k-i)) \} \)
- **Compute table of solutions:** \( M[1..n] \)
- **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 \( 1..n \)

**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 \( f[0]=f[1]=0 \) in FibDP...