Notes 11-1

Cameron Chalk

1 Memoization and iteration

In many naive recursive algorithms, redundant work is done in recursive calls. We can solve the problem by eliminating redundancy in one of two ways: memoization or iteration.

Consider the recursive function to generate the nth number in the Fibonacci sequence, fib(n):

- if $n = 1$ return 1
- if $n = 2$ return 1
- return fib($n - 2$) + fib($n - 1$)

This function does much redundant work; for example, when fib(7) is called, fib(6) and fib(5) are called. Importantly, when fib(6) is called, fib(5) is called again! Thus we are reevaluating the entire recursion which gives fib(5) at least twice, which is redundant. Try drawing a recursion tree for this function to see the immense redundancy for yourself. The runtime of this naive implementation is $\Theta(1.6\ldots^n)$[1].

To solve this redundancy issue and improve the runtime, we will memoize (i.e., write down) values of fib(n), so that we only evaluate them once:

```
fib(n):
    initialize array M of size n to all null
    M[1] = 1
    M[2] = 1
    Mfib(j):
        if M[j] = null
            M[j] := Mfib(j-2) + Mfib(j-1)
        endif
        return M[j]
    return Mfib(n)
```

[1]The nth Fibonacci number scales asymptotically as $\Theta(1.6\ldots^n)$. To understand why the runtime for the recursive algorithm also scales as $\Theta(1.6\ldots^n)$, try to first prove how many leaves there are in a recursion tree for fib(n), then prove how many total nodes there are, and note that the work done in each node is $\mathcal{O}(1)$.

To analyze the runtime, consider the following fact: to fill one entry in M, we must make at most two recursive calls. Thus, there are at most $2n$ recursive calls total, because when $M[i]$ is already filled, we do not make the recursive calls. Note that the algorithm Mfib does $\mathcal{O}(1)$ work, besides recursive calls. Then Mfib takes at most $2n \cdot \mathcal{O}(1) = \mathcal{O}(n)$. The lines before Mfib is called (initializing the array) also take $\mathcal{O}(n)$. Thus, our memoization technique reduces the runtime from exponential $\Theta(1.6\ldots^n)$ to linear time.

We can also compute Fibonacci iteratively:

```
fib(n)
    initialize array M of size n
    M[1] = 1
    M[2] = 2
    for i=3 to n
        M[i] = M[i-2] + M[i-1]
    return M[n]
```

Since we start from $M[3]$ and work towards n, we know that $M[i-1]$ and $M[i-2]$ are available when computing $M[i]$. The runtime is also $\mathcal{O}(n)$.

2 Dynamic programming

Dynamic programming is a technique which utilizes recursion and memoization/iterativity to solve problems which have slow runtimes when a naive recursive algorithm is used. The term dynamic programming is not very descriptive; this was done on purpose. In the 50's, Bellman was proposing the idea to Congress and asking for research funds. He coined dynamic programming as term that "not even a Congressman could object to". A more useful way to think of dynamic programming is recursion plus memoization/iteration.

See the Weighted Interval Scheduling slide on Canvas. We saw the Interval Scheduling problem in the greedy section, but the addition of weights to the jobs makes the Greedy algorithm suboptimal in many cases.

Towards a solution for the Weighted Interval Scheduling problem, we will differentiate between the following two concepts: the optimal solution (which jobs we choose) and **OPT**, the value of the optimal solution (the total value of the jobs we choose). See the next slide on Canvas titled Dynamic Programming: method for solving optimization problems. The slide shows a systematic way to solve many optimization problems.

Consider the Weighted Interval Scheduling problem. To find the recurrence for **OPT**, i.e., the optimal substructure, we want to reduce our problem into smaller subproblems. This is similar to Divide and Conquer algorithms, and the Fibonacci algorithm we saw above. Let J be the set of all jobs. Let $\mathbf{OPT}(J)$ be the value of the optimal solution for J. We want to define $\mathbf{OPT}(J)$ in terms of smaller instances of the Weighted Interval Scheduling problem.

Let's look at a particular job j. There are two cases: j either is or is not in the optimal solution. Suppose that we knew which case we were in; i.e.,

assume we know that j is not in the optimal solution. Then we can rewrite $\mathbf{OPT}(J) = \mathbf{OPT}(J \setminus \{j\})$, where $J \setminus \{j\}$ is the set of jobs J without job j).

In the other case, if we know j is in the optimal solution, we will use the following recurrence: $\mathbf{OPT}(J) = v_j + \mathbf{OPT}(J \setminus \{j \text{ and all jobs overlapping with } j\})\}$. We will see later why this is the right recurrence to use for this case.

The problem is, we do not know which case we are in; we do not know whether j will be in the optimal solution or not. Since we are trying to maximize **OPT**, we can do the following:

$$\mathbf{OPT}(J) = \max\{\mathbf{OPT}(J \setminus \{j\}, \tag{1}$$

$$v_j + \mathbf{OPT}(J \setminus \{j \text{ and all jobs overlapping with } j\}). \tag{2}$$

Claim 1. *Suppose $\{j_1, \ldots, j_k\}$ is an optimal solution for J. Then $\{j_1, \ldots, j_{k-1}\}$ is an optimal solution for $J' = J \setminus \{j$ and all jobs overlapping with $j\}$.*

Proof. The proof is an exchange argument, as we saw in the Greedy algorithms section. Suppose towards contradiction there is a solution $\{O_1, \ldots, O_m\}$ for J' which is more optimal than $\{j_1, \ldots, j_{k-1}\}$. Since none of the jobs in J' overlap with j_k, then $\{O_1, \ldots, O_m, j_k\}$ is a valid solution for J and has a higher value than $\{j_1, \ldots, j_k\}$ for J, which contradicts the assumption that $\{j_1, \ldots, j_k\}$ is optimal for J. □