Lecture 18

Greedy Algorithms: A *greedy algorithm* is one that when optimizing a function over some input, iteratively makes decisions by picking the decision that looks best at the moment. For example, when making change for n cents using coin denominations from the list L and trying to use as few coins as possible, we could keep picking the largest coin which is at most n cents until we've finally made change. Thus, even though overall the function we're trying to minimize is the total number of coins used, we at each step consider the biggest coin possible to be the best choice.

Sometimes greedy algorithms can provably produce optimal solutions, such as this example with US or Ethiopian currency (but as we saw earlier, this isn't always the case, e.g. making change for 8 cents with coin denominations $\{1, 4, 5\}$). Now we'll study a few more examples where greedy algorithms produce optimal solutions.

Hitting Intervals: Consider the following problem. The input is a list L of intervals, L = $[[a_1, b_1]$, $[a_2, b_2]$, ..., $[a_n, b_n]]$. These intervals might overlap. We would like to produce as small a set of numbers as possible so that each interval in the input contains at least one number in our set (in this case we say that each interval is *hit*). For example if the intervals were $[[1, 5]$, $[3, 7]]$, then producing the set $\{2, 6\}$ would hit both intervals, however the solution $\{4\}$ is better since it also hits both intervals and is a smaller set.

We first make one simple observation, which is that in the set we produce in the end, we only need to consider including numbers which are the right endpoint of some interval (proof: in any optimal solution, take each number in the set and keep increasing it until it *is* the right endpoint of some interval). Next, we also make the observation that if we only include right endpoints of intervals, then we *must* include the right endpoint of the interval with the smallest b_i, since that is the only way to hit that interval.

After the previous observations, we can develop an efficient algorithm for this problem via memoization. First sort L so that $b_1 \leq b_2 \leq \ldots \leq b_n$. Let $f(\text{at}, \text{last})$ be the minimum size of a set required to hit the intervals L[at],L[at+1],...,L[n] given that the last number we included in our set was L[last][1]. Then the optimal solution has value $1 + f(1, 0)$, since we have to include b_1 in our solution. In Python code:

```
def f(L, at, last, mem):
    if at == len(L):
        return 0
    elif mem[at][last] != -1:
        return mem[at][last]
    # include this interval's right endpoint in the output
    mem[at][last] = 1 + f(L, at+1, at, mem)
    if L[last][1]>=L[at][0] and L[last][1]<=L[at][1]:
        # don't include this interval's right endpoint in the output
        # but need to make sure this interval was already hit
        mem[at][last] = min(mem[at][last], f(L, at+1, last, mem))
    return mem[at][last]
```

```
def optimalHitting(L):
    # sort the list of intervals by their second coordinate
    for c in L:
        c[0],c[1] = c[1],c[0]
    L.sort()
    for c in L:
        c[0],c[1] = c[1],c[0]

    mem = []
    for i in xrange(len(L)):
        mem += [[-1]*len(L)]
    answer = 1 + f(L, 1, 0, mem)
    return answer
```

Sorting L in the beginning takes $O(n \log n)$ time via `mergeSort`. There are $n+1$ choices for `at` and n choices for `last`. The amount of work to fill in any particular entry of `mem` is $O(1)$, so in total the running time of the above implementation is $O(n^2)$.

By making one more observation, we can obtain an even faster *greedy* algorithm that runs in only $O(n \log n)$ time. First, what is the proposed greedy algorithm? We first sort L so that $b_1 \leq b_2 \leq \ldots \leq b_n$, taking $O(n \log n)$ time. We include b_1 in our solution then remove all intervals that have been hit to obtain a new list. Then we again include the smallest b_i remaining in our solution, and again remove all the intervals that are hit. We keep doing this until everything has been hit. In Python:

```
def optimalHitting(L):
    for c in L:
        c[0],c[1] = c[1],c[0]
    L.sort()
    for c in L:
        c[0],c[1] = c[1],c[0]

    ans = 1
    last = L[0][1]
    for x in L[1:]:
        if last>=x[0] and last<=x[1]:
            continue
        else:
            ans += 1
            last = x[1]
    return ans
```

Why does this produce the smallest output set possible? Well, as argued previously we can assume that our output set only contains right endpoints of intervals and must contain b_1. Now in our algorithm, we skip several intervals that are hit until we reach the first interval b_i which is not hit, and we include it. Suppose we had an optimal solution which actually took one or more of the b_j's that we skipped. We could take that solution, remove all such b_j's and replace them with b_i,

and we would have a new solution that is just as good or better and still hits every interval. By continuing in this way down the list, we can transform any solution into the greedy one in such a way that it can only improve.