Carlos Mc Gregor c.mcgregormuro@mail.utoronto.ca

CSC373S: Algorithm Design, Analysis & Complexity

Lecture 08

Monday January 23, 2017

based on notes by Denis Pankratov

Dynamic Programming

Weighted Interval Selection (Activity Selection)

Input: A - array of n activities, $A[i].s$ - start time, $A[i].f$ finish time, $A[i].w$ - weight
Output: $S \subseteq \{A[1], .., A[n]\}$ so that S compatible $\sum_{i \in S} A[i].w$ is as large as possible

EFT (*see Greedy Algorithm*) is no longer optimal (*see Figure 1 below*)

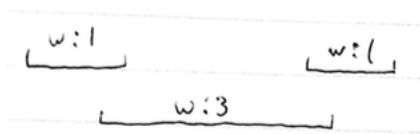

Figure 1: Failure of *Greedy Algorithm*

No known Greedy solution! Let OPT_n denote an optimal solution. Assume the activities are sorted by finishing times $A[1].f \leq A[2].f \leq \cdots \leq A[n].f$

Definition: prev$[j]$ = largest index i so that $A[i] \cdot f \leq A[j] \cdot s$.

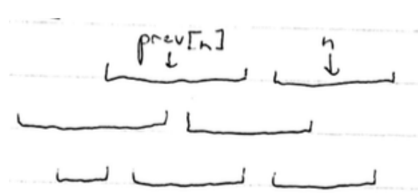

Figure 2: Visualization of weighted intervals

Still have optimal substructure. Two cases:

1. $A[n] \in OPT_n$, then OPT_n contains an optimal solution to the subproblem $A[1], \ldots A[\text{prev}[n]]$.
2. $A[n] \notin OPT_n$, then OPT_n contains an optimal solution to the subproblem $A[1], \ldots, A[n-1]$

Recursive algorithm for computing the value of OPT:

```
def ComputeOPT(A, j): # solve the subprolem A[1],..,A[j]
  if j = 0:
    return 0
  else:
    return max(A[j].w + ComputOPT(A, prev[j]),
                     ComputeOPT(A, j-1))
```

<u>Running Time:</u>

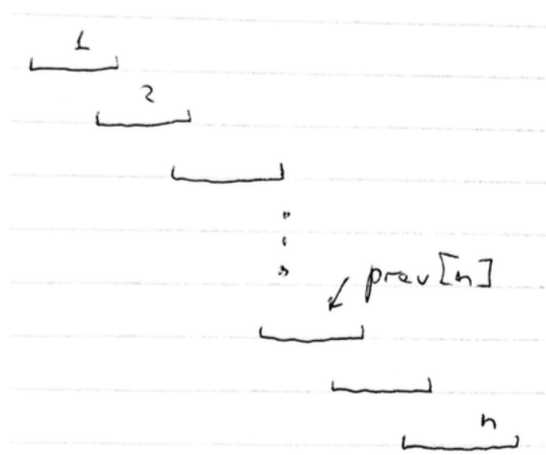

Figure 3: Example

Let $T(n)$ be the runtime on instances of size n.

$$T(n) = \begin{cases} \mathcal{O}(1), & \text{if } n = 0, 1 \\ T(n-1) + T(n-2), & \text{if } n > 1 \end{cases}$$

Fibonacci Type Recurrence: $T(n) = 2^{\Omega(n)}$

Overlapping Subproblems

<u>Solution</u>: store partial results into an array V

```
def ComputeOPT(A, V, j):
  if V[j] is not empty:
    return V[j]
  if j = 0:
    return V[0] = 0
```

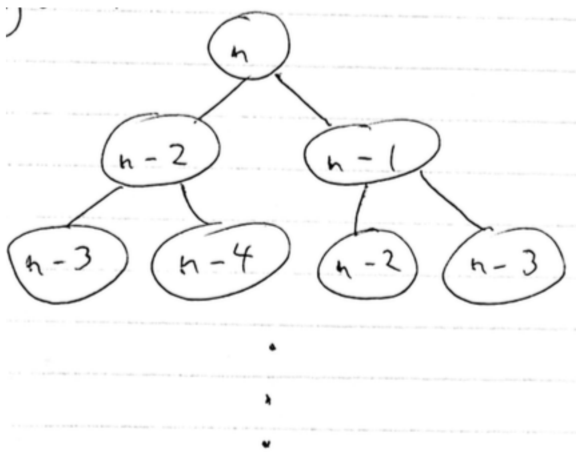

Figure 4: Visualization of overlapping subproblems

```
    else:
        return V[j] = max(A[j].w + ComputeOPT(A, V, prev[j]),
                          ComputeOPT(A, V, j - 1))
```

Running time becomes $\mathcal{O}(n)$.

Recursive solution + storing partial results is known as **MEMOIZATION**:

1. *Semantic Array*
   $V[j]$ = the value of optimal feasible subset of activities chosen from among:

   $$A[1], A[2], \ldots, A[j], 0 \leq j \leq n$$

   Answer is in $V[n]$.

2. *Computational Array*
   $V'[j] = \max(A[j].w + V[\text{prev}[j]], V'[j-1])$
   $V'[0] = 0 \leftarrow$ Base Case!

3. Correctness: Equivalence of the two arrays done before w/ the 2 cases

4. Algorithm:

```
def WISP(A, n):
    init V of size n + 1, 0-based indexing
    V[0] = 0
    for j = 1 to n:
        V[j] = max(A[j].w + V[prev[j]], V[j-1])

    return V[n]
```

5. Running Time: $\mathcal{O}(n)$ if A was already sorted and prev[] computed

Recover the actual set rather than the optimal value:

```
def WISPRecover(A, V, n):
  S = []
  k = n
  do:
    if V[j] = A[k].w + V[prev[k]]:
      S = S.append(k)
      k = prev[k]
    else:
      k -= 1
  while k > 0
```

O/I Knapsack Prob

Input: v - array of n values, w - array of n weights (positive integers), W - capactity (positive int)
Output: $S \subseteq [n]$ so that $\sum_{i \in S} W[i] \leq W$ and $\sum_{i \in S} V[i]$ is as large as possible

Exercise: Try greedy solutions, prove they *do not work*.

Semantic Array:
$C[i, k]$ = maximum value obtainable from items $1, .., i$ that fit into knapsack of capacity K, $0 \leq i \leq n, 0 \leq K \leq W$.

Answer is in $C[n, W]$.

Computational Array:

$$C[i, k] = \max \begin{cases} C[i-1, k] \\ V[i] + C[i-1, K - w[i]], & \text{if } w[i] \leq K \end{cases}$$

$$C[0, k] = 0, 0 \leq k \leq w$$

$$C[1, 0] = 0, 0 \leq i \leq n$$

Correctness pseudo-code: Exercise

Running Time: $\mathcal{O}(nW)$, not polynomial
Size of Input: $\mathcal{O}(n \cdot \max(\log V[i], log W[i]) + \log W)$ and $\sum_{i \in S} V[i]$ is as large as possible