COMS4231: Analysis of Algorithms

Fall 19

Alex Andoni

Lecture 9, 9/30/19

Last time

- Lower bound for sorting based on comparisons: $\Omega(n \log n)$
- Sorting beyond comparisons, in $O(n)$ time
  - Count Sort, Bucket Sort
  - Radix Sort

Today

- Dynamic Programming (new technique)
  - Fibonacci
  - 0-1 Knapsack
  - Longest Common Subsequence

Dynamic Programming

Reduce problem to smaller problems

like D&C but possibly *overlapping subproblems*

Memoization:

do not solve the same problem instance repeatedly,

solve it once and record the result to reuse it (if needed)

Bottom-up (iterative) version: Problems are solved from smaller to larger and solutions tabulated

Top-down (recursive) version: Before initiating recursive call, check if solution was already computed previously.

Example: Fibonacci numbers

- $F_0 = 0,\ F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}\ for\ n \geq 2$
- Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21,…

Recursive algorithm FIB(n):

if $n = 0$ **then return** 0

else if $n = 1$ **then return** 1

else return FIB($n - 1$) + FIB($n - 2$)

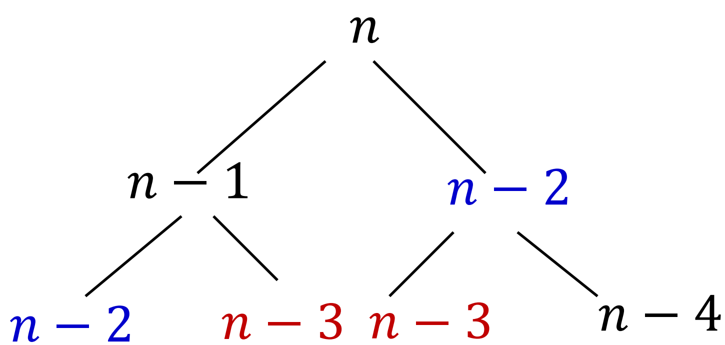
Complexity: $T(n) = T(n-1) + T(n-2) + O(1)$

Grows at least as fast as the Fibonacci numbers

$$F_n \approx \varphi^n / \sqrt{5},\ \text{where } \varphi = (1+\sqrt{5})/2 = \text{golden ratio}$$

Fibonacci numbers ctd.

- FIB is called only on n distinct arguments, but it is called repeatedly with the **same arguments**



Tabulate the result, so no need to evaluate it again on the same argument $\Rightarrow$ complexity $O(n)$

New algorithm FIB(n):

$F[0] = 0;\ F[1] = 1;$

for $i = 2$ **to** n **do**

$\quad F[i] = F[i-1] + F[i-2]$

return $F[n]$

(Actually, in this case we don't need an array: only need to remember the last two values)

often for Optimization Problems

Optimization Problem:

- For given instance of the problem, there is a set of "feasible" solutions, involving a number of choices (or decisions)
- Every solution has a cost or a value: metric for evaluating solutions
- Goal: find an optimal solution:
  - min-cost
  - or max-value

Dynamic Programming for Optimization Problems

Main principles:

1) Optimal Substructure:
Problem can be reduced to a set of smaller subproblems;
Optimal solution for whole involves optimal solutions for subproblems

2) Memoization:
Subproblems are solved from smaller to larger and solutions tabulated

Problem: 0-1 Knapsack

- n items, with given integer weights w_i, values v_i, $i = 1, \ldots, n$
- Knapsack with weight capacity W

- Goal: Choose a subset of items that fits in the knapsack and has maximum value

  i.e, choose a subset $S \subseteq \{1, \ldots, n\}$ that

  maximizes $\sum_{i \in S} v_i$

  subject to $\sum_{i \in S} w_i \leq W$

Example

Item	Weight	Value
1	4	25
2	6	30
3	2	10
4	5	27
5	7	35

Knapsack capacity: 13

Some feasible solutions, and their value

{1,2,3}: 65, {1,3,4}: 62, {1,3,5}: 70, {2,3,4}: 67,
{2,5}: 65, {4,5}: 62, ……

Optimal solution: {1,3,5}, value 70

Reduction to smaller subproblems: "last step/item analysis"

Should we take the n^{th} item?

- If we take it:

  we have capacity $W - w_n$ left and can pick any subset from the first $n-1$ items $\rightarrow$ ($n-1$ size subproblem)

- If we don't take it: we have capacity W for the first $n-1$ items $\rightarrow$ ($n-1$ size subproblem)

Overall optimum = the best of these 2 options!

Let $M(b,i)$ = maximum value we can get with knapsack of capacity b, using a subset of the first i items only

$M(b,i)$ = max{ $M(b-w_i, i-1) + v_i$, $M(b, i-1)$ } if $b \geq w_i$
else $M(b,i) = M(b,i-1)$
Base case: $M(b,0) = 0$ for all b

Recursive Algorithm (not D.P.)

$M(b,i)$ = max{ $M(b-w_i, i-1) + v_i$, $M(b, i-1)$ } if $b \geq w_i$
else $M(b,i) = M(b,i-1)$
Base case: $M(b,0) = 0$ for all b

Rec-KNAP(w,v,b,i)

[max value that can be obtained for capacity b from first i items only]

if $i = 0$ **or** $b = 0$ **then return** 0

else if $b < w_i$ **then return** Rec-KNAP$(w,v,b,i-1)$

else return max{ Rec-KNAP$(w,v,b-w_i,i-1) + v_i$, Rec-KNAP$(w,v,b,i-1)$ }

Main call: Rec-KNAP(w,v,W,n)

Time Complexity of Recursive algorithm

- A call for i items may generate two recursive calls with $i - 1$ items.

- $T(i) = 2T(i-1) + O(1)$
- $\Rightarrow T(n) = \Theta(2^n)$

- But: many of these recursive calls solve the same problem $M(b, i)$:

  at most nW different arguments.

  better if $W \ll 2^n$

DP Algorithm (iterative)

DP-KNAP(w, v, W)

for $b = 0$ **to** W **do** $M(b, 0) = 0$

for $i = 1$ **to** n **do**

  for $b = 0$ **to** W **do**

    if $b \geq w_i$ **and** $M(b - w_i, i - 1) + v_i > M(b, i - 1)$

      then $M(b, i) = M(b - w_i, i - 1) + v_i$

     else $M(b, i) = M(b, i - 1)$

Return $M(W, n)$

Running time: $O(nW)$

Ok if W is "small"

(strictly, not a *polynomial-time* algorithm if weights given in binary)

DP Algorithm: recursive version

Initialize M(b,i) = “?” at the beginning for all b,i

Main call: Rec-KNAP(w, v, W, n)

Rec-KNAP(w, v, b, i)

if $i = 0$ **or** $b = 0$ **then return** 0

if $M(b, i) \neq$ “?” **then return** $M(b, i)$

else if $b < w_i$ **then r=**Rec-KNAP$(w, v, b, i - 1)$

else r=max{ Rec-KNAP$(w, v, b - w_i, i - 1) + v_i$,
Rec-KNAP$(w, v, b, i - 1)$ }

$M(b, i) = \boldsymbol{r}$

Return r

Has same (worst-case) complexity as the iterative DP algorithm

Recovering an optimal solution

Record which case generates $M(b,i)$ for every b,i

for $b = 0$ **to** W **do** $M(b,0) = 0$

for $i = 1$ **to** n **do**

  for $b = 0$ **to** W **do**

    if $b \geq w_i$ **and** $M(b - w_i, i-1) + v_i > M(b, i-1)$

      then $\{M(b,i) = M(b - w_i, i-1) + v_i\ ;\ s(b,i) = 1\ \}$

    else $\{\ M(b,i) = M(b, i-1)\ ;\ s(b,i) = 0\ \}$

Return $M(W,n)$ and s

$s(b,i) = 1$ iff (an) optimal solution for $M(b,i)$ uses item i

Optimal Solution:

$b = W;\ S = \emptyset$

for $i = n$ **down to** 1 **do**

  if $s(b,i) = 1$ **then** $\{\ S = S \cup \{i\};\ b = b - w_i\ \}$

return S

Longest Common Subsequence (LCS)

- Given two sequences $x[1 \ldots m], y[1 \ldots n]$, find a longest common subsequence (gaps allowed)

x = A T C T T A G

y = T G C A T A

- Applications: computational biology, *diff*

- Naive way: Take every subsequence of x check against y $\rightarrow$ Time: $2^m n$