Algorithms

Dynamic Programming

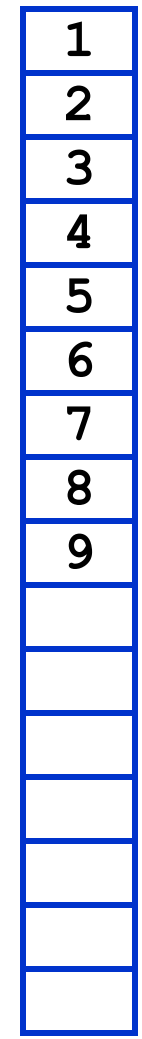
Review: Amortized Analysis

- To illustrate amortized analysis we examined *dynamic tables*
  1. Init table size $m = 1$
  2. Insert elements until number $n > m$
  3. Generate new table of size $2m$
  4. Reinsert old elements into new table
  5. (back to step 2)
- *What is the worst-case cost of an insert?*
- *What is the amortized cost of an insert?*

Review: Analysis Of Dynamic Tables

- Let c_i = cost of *i*th insert
- $c_i = i$ if i-1 is exact power of 2, 1 otherwise
- Example:
  - Operation / Table Size / Cost

Operation	Table Size	Cost
`Insert(1)`	1	1
`Insert(2)`	2	1 + 1
`Insert(3)`	4	1 + 2
`Insert(4)`	4	1
`Insert(5)`	8	1 + 4
`Insert(6)`	8	1
`Insert(7)`	8	1
`Insert(8)`	8	1
`Insert(9)`	16	1 + 8



Review: Aggregate Analysis

- n Insert() operations cost

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lg n} 2^j = n + (2n - 1) < 3n$$

- Average cost of operation
= (total cost)/(# operations) < 3
- Asymptotically, then, a dynamic table costs the same as a fixed-size table
  - Both O(1) per Insert operation

Review: Accounting Analysis

- Charge each operation \$3 amortized cost
  - Use \$1 to perform immediate Insert()
  - Store \$2
- When table doubles
  - \$1 reinserts old item, \$1 reinserts another old item
  - We've paid these costs up front with the last *n*/2 Insert()s
- Upshot: O(1) amortized cost per operation

Review: Accounting Analysis

- Suppose must support insert & delete, table should contract as well as expand
  - Table overflows ⇒ double it (as before)
  - Table < 1/4 full ⇒ halve it
  - Charge \$3 for Insert (as before)
  - Charge \$2 for Delete
    - Store extra \$1 in emptied slot
    - Use later to pay to copy remaining items to new table when shrinking table
- *What if we halve size when table < 1/8 full?*

Dynamic Programming

- Another strategy for designing algorithms is *dynamic programming*
  - A metatechnique, not an algorithm (like divide & conquer)
  - The word “programming” is historical and predates computer programming
- Use when problem breaks down into recurring small subproblems

Dynamic Programming Example: Longest Common Subsequence

- *Longest common subsequence* (*LCS*) problem:
  - Given two sequences x[1..m] and y[1..n], find the longest subsequence which occurs in both
  - Ex: x = {A B C B D A B }, y = {B D C A B A}
  - {B C} and {A A} are both subsequences of both
    - *What is the LCS?*
  - Brute-force algorithm: For every subsequence of x, check if it's a subsequence of y
    - *How many subsequences of x are there?*
    - *What will be the running time of the brute-force alg?*

LCS Algorithm

- Brute-force algorithm: 2^m subsequences of x to check against *n* elements of y: $O(n\,2^m)$
- We can do better: for now, let's only worry about the problem of finding the *length* of LCS
  - When finished we will see how to backtrack from this solution back to the actual LCS
- Notice LCS problem has optimal substructure
  - Subproblems: LCS of pairs of *prefixes* of x and y

Finding LCS Length

- Define c[*i*,*j*] to be the length of the LCS of x[1..*i*] and y[1..*j*]
  - *What is the length of LCS of x and y?*
- Theorem:

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- *What is this really saying?*