Lecture 25: AlgoRhythm Design Techniques

✦ Agenda for today’s class:
  ➤ Coping with NP-complete and other hard problems
    • Approximation using Greedy Techniques
      • Optimally bagging groceries: Bin Packing
    • Divide & Conquer Algorithms and their Recurrences
    • Dynamic Programming by “memoizing”
      • Fibonacci’s Revenge
    • Randomized Data Structures and Algorithms
      • Treaps
      • “Probably correct” primality testing
  ➤ In the Sections on Thursday: Backtracking
    • Game Trees, minimax, and alpha-beta pruning

✦ Read Chapter 10 and Sec 12.5 in the textbook

Recall: P, NP, and Exponential Time Problems

✦ Diagram depicts relationship between P, NP, and EXPTIME
  (class of problems that can be solved within exponential time)

✦ NP-Complete problem = problem in NP to which all other NP problems can be reduced
  ➤ Can convert input for a given NP problem to input for NPC problem

✦ All algorithms for NP-C problems so far have tended to run in nearly exponential worst case time

It is believed that
$P \neq NP \neq EXPTIME$
The “Curse” of NP-completeness

✦ Cook first showed (in 1971) that satisfiability of Boolean formulas (SAT) is NP-Complete

✦ Hundreds of other problems (from scheduling and databases to optimization theory) have since been shown to be NPC

✦ No polynomial time algorithm is known for any NPC problem!

Coping strategy #1: Greedy Approximations

✦ Use a greedy algorithm to solve the given problem
  ➞ Repeat until a solution is found:
    ♦ Among the set of possible next steps:
      Choose the current best-looking alternative and commit to it

✦ Usually fast and simple

✦ Works in some cases…(always finds optimal solutions)
  ➞ Dijsktra’s single-source shortest path algorithm
  ➞ Prim’s and Kruskal’s algorithm for finding MSTs

✦ but not in others…(may find an approximate solution)
  ➞ TSP – always choosing current least edge-cost node to visit next
  ➞ Bagging groceries…
The Grocery Bagging Problem

✦ You are an environmentally-conscious grocery bagger at QFC
✦ You would like to minimize the total number of bags needed to pack each customer’s items.

Items (mostly junk food)  Grocery bags

Sizes \( s_1, s_2, \ldots, s_N \) (0 < \( s_i \leq 1 \))  Size of each bag = 1

Optimal Grocery Bagging: An Example

✦ Example: Items = 0.5, 0.2, 0.7, 0.8, 0.4, 0.1, 0.3
✦ How many bags of size 1 are required?

0.2 0.3 0.1
0.8 0.7 0.4
0.5

Only 3 bags required

✦ Can find optimal solution through exhaustive search
✦ Search all combinations of N items using 1 bag, 2 bags, etc.
✦ Takes exponential time!
Bagging groceries is NP-complete

- **Bin Packing problem**: Given \(N\) items of sizes \(s_1, s_2, \ldots, s_N\) (\(0 < s_i \leq 1\)), pack these items in the least number of bins of size 1.

![Diagram of items and bins](image)

Sizes \(s_1, s_2, \ldots, s_N\) (\(0 < s_i \leq 1\))  
Size of each bin = 1

- The general bin packing problem is NP-complete
  - Reductions: All NP-problems → SAT → 3SAT → 3DM → PARTITION → Bin Packing (see Garey & Johnson, 1979)

Greedy Grocery Bagging

- Greedy strategy #1 "**First Fit**":
  1. Place each item in first bin large enough to hold it
  2. If no such bin exists, get a new bin

- **Example**: Items = 0.5, 0.2, 0.7, 0.8, 0.4, 0.1, 0.3
Greedy Grocery Bagging

- Greedy strategy #1 “First Fit”:
  1. Place each item in first bin large enough to hold it
  2. If no such bin exists, get a new bin

- Example: Items = 0.5, 0.2, 0.7, 0.8, 0.4, 0.1, 0.3

- Approximation Result: If M is the optimal number of bins, First Fit never uses more than \[\lceil 1.7M \rceil\] bins (see textbook).

Getting Better at Greedy Grocery Bagging

- Greedy strategy #2 “First Fit Decreasing”:
  1. Sort items according to decreasing size
  2. Place each item in first bin large enough to hold it

- Example: Items = 0.5, 0.2, 0.7, 0.8, 0.4, 0.1, 0.3
Getting Better at Greedy Grocery Bagging

✦ Greedy strategy #2 “First Fit Decreasing”:
  1. Sort items according to decreasing size
  2. Place each item in first bin large enough to hold it

✦ Example: Items = 0.5, 0.2, 0.7, 0.8, 0.4, 0.1, 0.3

<table>
<thead>
<tr>
<th>Item 1</th>
<th>Item 2</th>
<th>Item 3</th>
<th>Item 4</th>
<th>Item 5</th>
<th>Item 6</th>
<th>Item 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>0.5</td>
<td>0.1</td>
<td>0.7</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Uses 3 bins
Optimal in this case
Not optimal in general

✦ Approximation Result: If M is the optimal number of bins, First Fit Decreasing never uses more than 1.2M + 4 bins (see textbook).

Coping Strategy #2: Divide and Conquer

✦ Basic Idea:
  1. Divide problem into multiple smaller parts
  2. Solve smaller parts (“divide”)
     ♦ Solve base cases directly
     ♦ Solve non-base cases recursively
  3. Merge solutions of smaller parts (“conquer”)

✦ Elegant and simple to implement
  ✤ E.g. Mergesort, Quicksort, etc.

✦ Run time T(N) analyzed using a recurrence relation:
  ✤ T(N) = aT(N/b) + Θ(N^k) where a ≥ 1 and b > 1
Analyzing Divide and Conquer Algorithms

- Run time T(N) analyzed using a recurrence relation:
  \[ T(N) = aT(N/b) + \Theta(N^k) \]
  where \( a \geq 1 \) and \( b > 1 \)

- General solution (see theorem 10.6 in text):

\[
T(N) = \begin{cases} 
O(N^{\log_b a}) & \text{if } a > b^k \\
O(N^k \log N) & \text{if } a = b^k \\
O(N^k) & \text{if } a < b^k 
\end{cases}
\]

- Examples:
  - Mergesort: \( a = b = 2, k = 1 \) \( \Rightarrow T(N) = O(N \log N) \)
  - Three parts of half size and \( k = 1 \) \( \Rightarrow T(N) = O(N^{\log_2 3}) = O(N^{1.59}) \)
  - Three parts of half size and \( k = 2 \) \( \Rightarrow T(N) = O(N^2) \)

Another Example of D & C

- Recall our old friend Signor Fibonacci and his numbers:

\[ 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots \]

- First two are: \( F_0 = F_1 = 1 \)
- Rest are sum of preceding two
  \[ F_n = F_{n-1} + F_{n-2} \text{ (} n > 1 \text{)} \]
A D & C Algorithm for Fibonacci Numbers

- public static int fib(int i) {
  if (i < 0) return 0; // invalid input
  if (i == 0 || i == 1) return 1; // base cases
  else return fib(i-1)+fib(i-2);
}

- Easy to write: looks like the definition of $F_n$
- But what is the running time $T(N)$?

Recursive Fibonacci

- public static int fib(int N) {
  if (N < 0) return 0; // time = 1 for the < operation
  if (N == 0 || N == 1) return 1; // time = 3 for 2 ==, 1 ||
  else return fib(N-1)+fib(N-2); // $T(N-1)+T(N-2)+1$
}

- Running time $T(N) = T(N-1) + T(N-2) + 5$
- Using $F_n = F_{n-1} + F_{n-2}$ we can show by induction that $T(N) \geq F_N$.
- We can also show by induction that $F_N \geq (3/2)^N$
Recursive Fibonacci

- public static int fib(int N) {
  if (N < 0) return 0; // time = 1 for the < operation
  if (N == 0 || N == 1) return 1; // time = 3 for 2 ==, 1 ||
  else return fib(N-1)+fib(N-2); // T(N-1)+T(N-2)+1
}
- Running time $T(N) = T(N-1) + T(N-2) + 5$
- Therefore, $T(N) \geq (3/2)^N$
  i.e. $T(N) = \Omega((1.5)^N)$

Yikes…exponential running time!

The Problem with Recursive Fibonacci

- Wastes precious time by re-computing $fib(N-i)$ over and over again, for $i = 2, 3, 4,$ etc.!
Solution: “Memoizing” (Dynamic Programming)

✦ Basic Idea: Use a table to store subproblem solutions
  ➢ Compute solution to a subproblem only once
  ➢ Next time the solution is needed, just look-up the table

✦ General Structure of DP algorithms:
  ➢ Define problem in terms of smaller subproblems
  ➢ Solve & record solution for each subproblem & base cases
  ➢ Build solution up from solutions to subproblems

Memoized (DP-based) Fibonacci

✦ public static int fib(int i) {
    // create a global array fibs to hold fib numbers
    // int fibs[N]; // Initialize array fibs to 0’s
    if (i < 0) return 0; // invalid input
    if (i == 0 || i == 1) return 1; // base cases
    // compute value only if previously not computed
    if (fibs[i] == 0)
        fibs[i] = fib(i-1)+fib(i-2); // update table (memoize!)
    return fibs[i];
}

Run Time = ?
The Power of DP

- Each value computed only once! No multiple recursive calls
- N values needed to compute $\text{fib}(N)$  \hspace{1cm} \text{Run Time} = O(N)

Summary of Dynamic Programming

- Very important technique in CS: Improves the run time of D & C algorithms whenever there are shared subproblems
- Examples:
  - DP-based Fibonacci
  - Ordering matrix multiplications
  - Building optimal binary search trees
  - All-pairs shortest path
  - DNA sequence alignment
  - Optimal action-selection and reinforcement learning in robotics
  - etc.
Coping Strategy #3: Viva Las Vegas! (Randomization)

✦ Basic Idea: When faced with several alternatives, toss a coin and make a decision
  ⇒ Utilizes a pseudorandom number generator (Sec. 10.4.1 in text)

✦ Example: Randomized QuickSort
  ⇒ Choose pivot randomly among array elements

✦ Compared to choosing first element as pivot:
  ⇒ Worst case run time is $O(N^2)$ in both cases
    ♦ Occurs if largest chosen as pivot at each stage
  ⇒ BUT: For same input, randomized algorithm most likely won’t repeat bad performance whereas deterministic quicksort will!
  ⇒ Expected run time for randomized quicksort is $O(N \log N)$ time for any input

Randomized Data Structures

✦ We’ve seen many data structures with good average case performance on random inputs, but bad behavior on particular inputs
  ⇒ E.g. Binary Search Trees

✦ Instead of randomizing the input (which we cannot!), consider randomizing the data structure!
What’s the Difference?

✦ Deterministic data structure with good average time over all inputs
  ❗️ If your application happens to always contain the “bad” inputs, you are in big trouble!

✦ Randomized data structure with good expected time for any input
  ❗️ Once in a while you will have an expensive operation, but no input can make this happen all the time

✦ Kind of like an insurance policy for your algorithm!

(Disclaimer: Allstate wants nothing to do with this boring lecture or lecturer.)
Example: Treaps (= Trees + Heaps)

- Treaps have both the binary search tree property as well as the heap-order property.
- Two keys at each node:
  - Key 1 = search element
  - Key 2 = randomly assigned priority

Legend:
- Priority
- Search key

Treap Insert

- Create node and **assign it a random priority**
- Insert as in normal BST
- **Rotate up until heap order is restored** (while maintaining BST property)
Why Bother?

Tree + Heap...

✦ Inserting sorted data into a BST gives poor performance!
✦ Try inserting data in sorted order into a treap. What happens?

insert(7)  insert(8)  insert(9)  insert(12)

6
7

7
8

6
7

2
9

6
7

2
9

7
8

7
8

7
8

15
12

Tree shape does not depend on input order anymore!

Treap Summary

✦ Implements (randomized) Binary Search Tree ADT
  ➢ Insert in expected $O(\log N)$ time for any input
  ➢ Delete in expected $O(\log N)$ time for any input
    ✦ Find the key and increase its value to $\infty$
    ✦ Rotate it to the fringe
    ✦ Snip it off
  ➢ Find in expected $O(\log N)$ time for any input
  ➢ But worst case is $O(N)$

✦ Memory use
  ➢ $O(1)$ per node
  ➢ About the cost of AVL trees

✦ Very simple to implement, little overhead
  ➢ Unlike AVL trees, no need to update balance information!
Final Example: Randomized Primality Testing

✦ Problem: Given a number \( N \), is \( N \) prime?

✦ Important for cryptography

✦ Randomized Algorithm based on a Result by Fermat:
  1. Guess a random number \( A, 0 < A < N \)
  2. If \( (A^{N-1} \mod N) \neq 1 \), then Output “\( N \) is not prime”
  3. Otherwise, Output “\( N \) is (probably) prime”
     – \( N \) is prime with high probability but not 100%
     – \( N \) could be a “Carmichael number” – a slightly more complex test rules out this case (see text)
     – Can repeat steps 1-3 to make error probability close to 0

✦ Recent breakthrough: Polynomial time algorithm that is always correct (runs in \( O(\log^{12} N) \) time for input \( N \))


To Do:
Read Chapter 10 and Sec. 12.5 (treaps)
Finish HW assignment #5

Next Time:
A Taste of Amortization
Final Review