The Problem with Deterministic Data Structures

We’ve seen many data structures with good average case performance on random inputs, but bad behavior on particular inputs.

We define the worst case runtime over all possible inputs $I$ of size $n$ as:

$$\text{Worst-case } T(n) = \max_I T(I)$$

We define the average case runtime over all possible inputs $I$ of size $n$ as:

$$\text{Average-case } T(n) = \frac{\sum_I T(I)}{\text{numPossInputs}}$$

The Motivation for Randomization

Instead of randomizing the input (since we cannot!), consider randomizing the data structure.

- No bad inputs, just unlucky random numbers
- Expected case good behavior on any input

Expectant Cases

Definition:

- A worst-case expected time analysis is a weighted sum of all possible outcomes over some probability distribution.

Thus, the expected runtime of a randomized data structure on some input $I$ is:

$$\text{Expected } T(I) = \sum_S \text{Pr}(S) \cdot T(I, S)$$

And the worst-case expected runtime of a randomized data structure* is:

$$\text{Expected } T(n) = \max_S \left( \sum_S \text{Pr}(S) \cdot T(I, S) \right)$$

* Randomized data structure = a data structure whose behaviour is dependant on a sequence of random numbers

What’s the Difference?

- Randomized with good expected time:
  - Once in a while you will have an expensive operation, but no inputs can make this happen all the time.

- Deterministic with good average time:
  - If your application happens to always use the “bad” case, you are in big trouble!

- Expected time is kind of like an insurance policy for your algorithm!
Does Randomization Work?
- Not-really randomized data structures
  - Splay Trees (compare with AVL trees)
  - Skew Heaps (compare with Leftist heaps)
  - Others?
- Randomized data structures
  - Universal Hashing hash table
    - Also Perfect Hashing hash table
  - Others?

Treap Data Structure for the Dictionary ADT

Treaps:
- Have the binary tree \textit{structure} property
- Have the BST \textit{order} property
- Have the heap order property with \textit{randomly} assigned priorities

Legend:
- heap in yellow; search tree in blue

Treap Insert
- Choose a random priority
- Insert as in normal BST
- Rotate up until heap order is restored (maintaining BST property while rotating)

Treap Delete
- Find the key
- Increase its priority to $\infty$
- Rotate it to the fringe
- Snip it off

Tree + Heap… Why Bother?

Insert data in sorted order into a treap; what shape tree comes out?

Treap Delete, cont.
Treap Summary

Implements Dictionary ADT
- Insert in expected $O(\log n)$ time
- Delete in expected $O(\log n)$ time
- Find in expected $O(\log n)$ time
- But worst case $O(n)$

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

Very simple to implement, little overhead
- Less than AVL trees

Perfect Binary Skip List

- Sorted linked list
- # of links of a node is its height
- The height $i$ link of each node (that has one) links to the next node of height $i$ or greater

Find() in a Perfect Binary Skip List

- Start $i$ at the maximum height
- Until the node is found, or $i=1$ and the next node is too large:
  - If the next node along the $i$ link is less than the target, traverse to the next node
  - Otherwise, decrease $i$ by one

Insert() in a Perfect Binary Skip List

Randomized Skip List Intuition

- It’s far too hard to insert into a perfect skip list
- But is perfection necessary?
- What matters in a skip list?
  - We want way fewer tall nodes than short ones
  - Make good progress through the list with each high traverse

Randomized Skip List

- Sorted linked list
- # of links of a node is its height
- The height $i$ link of each node (that has one) links to the next node of height $i$ or greater
- There should be about $1/2$ as many height $i+1$ nodes as height $i$ nodes
Find() in a RSL

- Start \( i \) at the maximum height
- Until the node is found or \( i \) is one and the next node is too large:
  - If the next node along the \( i \) link is less than the target, traverse to the next node
  - Otherwise, decrease \( i \) by one

Same as for a perfect skip list!

Insert() in a RSL

- Flip a coin until it comes up heads
  - This will take \( i \) flips. Make the new node’s height \( i \).
- Do a find, remembering nodes where we moved down one link
- Add the new node at the spot where the find ends
- Point all the nodes where we moved down (up to the new node’s height) at the new node
- Point the new node’s links where those redirected pointers were pointing

RSL Insert Example

```
2
19
23
8
13
29
20
10
22
```

Iteration and 1D Range Queries

- Iteration: successively return (in order) each element in the structure
  - Start at beginning, walk list to end
  - Just like a linked list!
- Range query: search for everything that falls between two values
  - Find() start point
  - Walk through skip list using the lowest links
  - Output each node until the end point

Randomized Skip List

Summary

- Implements Dictionary ADT
  - Insert in expected \( O(\log n) \)
  - Find in expected \( O(\log n) \)
  - But worst case \( O(n) \)
- Memory use
  - \( O(1) \) memory per node
  - About double a linked list
- Less overhead than search trees for iteration over range queries

Intermission: Efficiently Calculating Powers

- How would you implement a function/method \( \text{pow}(x, n) \) which returns the number \( x^n \)?
- How could you do that efficiently?
Primality Checking

- Given a number \( P \), can we determine whether or not \( P \) is prime?

Date: Wed, 7 Aug 2002 11:00:43 -0700 (PDT)
Newsgroups: cs-cs.ugrats.openforum
Subject: Primes in P??

So, a paper published yesterday alleges they have found a deterministic polynomial algorithm to determine primality.

http://www.cse.iitk.ac.in/primality.pdf

Two Properties of Primes

- \( P \) is a prime \( 0 < A < P \) and \( 0 < X < P \)

Then:
\[
A^{P-1} = 1 \pmod{P}
\]

And, the only solutions to \( X^2 = 1 \pmod{P} \) are:

\[
X = 1 \text{ and } X = P - 1
\]

Checking Primality - First Attempt

```c
int pow(int a, int n) {
    if (n == 0)
        return 1;
    if (n == 1)
        return a;
    int x = pow(a, n/2);
    if( isEven(n) )
        return x * x;
    else
        return x * x * a;
}
```

Checking Primality - Second Attempt

```c
int pow(int a, int n, int p) {
    if (n == 0)
        return 1;
    if (n == 1)
        return a;
    int x = pow(a, n/2, p);
    int squared = x * x % p;
    if( squared == 1 && x is neither p-1 or 1)
        return x * x;
    else
        return x * x * a;
}
```

Using More Information

“And, the only solutions to \( X^2 = 1 \pmod{P} \) are:

\[
X = 1 \text{ and } X = P - 1
\]”

Checking Primality

Systematic algorithm:
- For all \( A \) such that \( 0 < A < P \)
- Calculate \( A^{P-1} \pmod{P} \) using \( \text{pow}() \)
- Check at each step of \( \text{pow}() \) and at end for primality conditions

Randomized algorithm:
- Randomly pick an \( A \) and calculate \( A^{P-1} \pmod{P} \) using \( \text{pow}() \)

If \( \text{pow}() \) tells us that \( P \) is prime, will either algorithm tell us conclusively that \( P \) really is prime?
Randomized Primality Check

If the randomized algorithm reports failure, then \( P \) really isn’t prime.

If the randomized algorithm reports success, then \( P \) might be prime.

- \( P \) is prime with probability \( > \frac{3}{4} \)
- Each new \( A \) has independent probability of false positive

Solution:
- Run the randomized algorithm several times

Evaluating Randomized Primality Testing

Your probability of being struck by lightning this year: 0.00004%

Your probability that a number that tests prime 11 times in a row is actually not prime: 0.00003%

Your probability of winning a lottery of 1 million people five times in a row: 1 in \( 2^{100} \)

Your probability that a number that tests prime 50 times in a row is actually not prime: 1 in \( 2^{100} \)

Randomized Graph Searching

Consider some really huge graphs…

- All cities and towns in the World Atlas
- All stars in the Galaxy
- All ways 10 blocks can be stacked

Huh???

Implicitly Generated Graphs

- A huge graph may be implicitly specified by rules for generating it on-the-fly
- Blocks world:
  - vertex = relative positions of all blocks
  - edge = robot arm stacks one block

The Blocks World Problem:
A Large Branching Factor

- Source = initial state of the blocks
- Goal = desired state of the blocks
- Path source to goal = sequence of actions (program) for robot arm!
- \( n \) blocks = \( n^3 \) vertices
  - 10 blocks = 10 billion vertices!
- We cannot search such huge graphs exhaustively!
  - Breadth-first search: If out-degree of each node is 10, potentially visits \( 10^n \) vertices
  - Dijkstra’s algorithm is basically breadth-first search (modified to handle edge weights)

Review: Heuristic-Based Searching

- The Manhattan distance (\( \Delta x + \Delta y \)) is an estimate of the distance to the goal
  - A heuristic value
- Best-First Search
  - Select nodes to minimize estimated distance to the goal; if a promising set of nodes doesn’t pan out, backtrack
- Hill-climbing
  - Select nodes to minimize estimated distance to the goal
  - Says nothing about backtracking. In fact, we don’t even keep track of our previous path!
“Hill Climbing”: What’s in a Name?

- Let’s assume we’re trying to maximize our heuristic
- If we view our graph as a terrain, and the heuristic values as elevations, then graph searching becomes a problem of finding the tallest hill
- But …
  - What are some problems with hill climbing?

Solution: Hill Climbing with Random Restarts*

- Once you can’t go any farther, randomly choose a node in the graph, and try again.
- Once you have several possible solutions, pick the one with the highest heuristic value
- A common variation is simulated annealing, which may pick a random move instead of the best move. As the algorithm progresses, we choose the random move less and less often

Example: N-Queens Problem

- Place N queens on an N by N chessboard so that no two queens can attack each other
- Graph search formulation:
  - Each way of placing from 0 to N queens on the chessboard is a vertex
  - Edge between vertices that differ by adding or removing one queen
  - Start vertex: empty board
  - Goal vertex: any one with N non-attacking queens (there are many such goals)

Hill Climbing with Random Restarts – Complexity?

- Can often prove that if you run long enough will reach a goal state – but may take exponential time
- In some cases can prove that a hill-climbing or random walk algorithm will find a goal in polynomial time with high probability
  - e.g., 2-SAT, Papadimitriou 1997
- Widely used for real-world problems where actual complexity is unknown – scheduling, optimization
  - N-Queens – probably polynomial, but no one has tried to prove formal bound

Other Real-World Applications

- Routing finding – computer networks, airline route planning
- VLSI layout – cell layout and channel routing
- Production planning – “just in time” optimization
- Protein sequence alignment
- Travelling Salesman
- Many other “NP-Hard” problems
  - A class of problems for which no exact polynomial time algorithms exist – so heuristic search is the best we can hope for

Other Randomized Algorithms/Data Structures

- Data Structures
  - Other Dictionary ADT’s
- Algorithms
  - Find a minimum spanning tree in $O(E)$ time
  - Max flow problems in $O(V^2 \log V)$ time
    - In CSE 421, you’ll see a deterministic algorithm works in $O(V^3)$ time
  - Many others!