Hashing: Log (10^{100}) is a big number

- Comparison based searches are too slow for lots of data
  - How many comparisons needed for a billion elements?
  - What if one billion web-pages indexed?

- Hashing is a search method: average case O(1) search
  - Worst case is very bad, but in practice hashing is good
  - Associate a number with every key, use the number to store the key
    - Like catalog in library, given book title, find the book

- A hash function generates the number from the key
  - Goal: Efficient to calculate
  - Goal: Distributes keys evenly in hash table

Hashing details

- There will be collisions, two keys will hash to the same value
  - We must handle collisions, still have efficient search
  - What about birthday "paradox": using birthday as hash function, will there be collisions in a room of 25 people?

- Several ways to handle collisions, in general array/vector used
  - Linear probing, look in next spot if not found
    - Hash to index h, try h+1, h+2, ..., wrap at end
    - Clustering problems, deletion problems, growing problems
  - Quadratic probing
    - Hash to index h, try h+1^2, h+2^2, h+3^2, ..., wrap at end
    - Fewer clustering problems
  - Double hashing
    - Hash to index h, with another hash function to j
    - Try h, h+j, h+2j, ...

Chaining with hashing

- With n buckets each bucket stores linked list
  - Compute hash value h, look up key in linked list table[h]
  - Hopefully linked lists are short, searching is fast
  - Unsuccessful searches often faster than successful
    - Empty linked lists searched more quickly than non-empty
  - Potential problems?

- Hash table details
  - Size of hash table should be a prime number
  - Keep load factor small: number of keys/size of table
  - On average, with reasonable load factor, search is O(1)
  - What if load factor gets too high? Rehash or other method

Hashing problems

- Linear probing, hash(x) = x, (mod tablesize)
  - Insert 24, 12, 45, 14, delete 24, insert 23 (where?)

- Same numbers, use quadratic probing (clustering better?)

- What about chaining, what happens?
What about hash functions

- Hashing often done on strings, consider two alternatives

```java
public static int hash(String s) {
    int k, total = 0;
    for(k=0; k < s.length(); k++) {
        total += s.charAt(k);
    }
    return total;
}
```

- Consider `total += (k+1)*s.charAt(k)`, why might this be better?
  - Other functions used, *always mod result by table size*

- What about hashing other objects?
  - Need conversion of key to index, not always simple
  - Every object has method `hashCode()`!

Tools: Solving Computational Problems

- Algorithmic techniques and paradigms
  - Brute-force/exhaustive, greedy algorithms, dynamic programming, divide-and-conquer, ...
  - Transcend a particular language
  - Designing *algorithms*, may change when turned into code

- Programming techniques and paradigms
  - Recursion, memo-izing, compute-once/lookup, tables, ...
  - Transcend a particular language
  - Help in making code work
    - Avoid software problems (propagating changes, etc.)
    - Avoid performance problems

Quota Exceeded

- You’re running out of disk space
  - Buy more
  - Compress files
  - Delete files

- How do you find your “big” files?
  - What’s big?
  - How do you do this?

Recursive structure matches code

```java
public static long THRESHOLD = 1000000L; // one million

public static void findBig(File dir, String tab) {
    File[] dirContents = dir.listFiles();
    System.out.println(tab+dir.getPath());
    for(File f : dirContents) {
        if (f.isDirectory()) {
            findBig(f, tab+"	");
        } else {
            if (f.length() > THRESHOLD) {
                System.out.printf("%s%s%8d
", tab,f.getName(), f.length());
            }
        }
    }
}
```
Solving Problems Recursively

- Recursion is an indispensable tool in a programmer's toolkit
  - Allows many complex problems to be solved simply
  - Elegance and understanding in code often leads to better programs: easier to modify, extend, verify (and sometimes more efficient!!)
  - Sometimes recursion isn’t appropriate, when it’s bad it can be very bad—every tool requires knowledge and experience in how to use it

- The basic idea is to get help solving a problem from coworkers (clones) who work and act like you do
  - Ask clone to solve a simpler but similar problem
  - Use clone’s result to put together your answer
- Need both concepts: call on the clone and use the result

Print words read, but print backwards

- Could store all the words and print in reverse order, but ...
  - Probably the best approach, recursion works too

```java
public void printReversed(Scanner s){
    if (s.hasNext()){ // reading succeeded?
        String word = s.next(); // store word
        printReversed(s); // print rest
        System.out.println(word); // print the word
    }
}
```

- The function printReversed reads a word, prints the word only after the clones finish printing in reverse order
  - Each clone has own version of the code, own word variable
  - Who keeps track of the clones?
  - How many words are created when reading N words?
    - Can we do better?

Exponentiation

- Computing \(x^n\) means multiplying \(n\) numbers (or does it?)
  - What's the simplest value of \(n\) when computing \(x^n\)?
  - If you want to multiply only once, what can you ask a clone?

```java
public static double power(double x, int n){
    if (n == 0){
        return 1.0;
    }
    return x * power(x, n-1);
}
```

- Number of multiplications?
  - Note base case: no recursion, no clones
  - Note recursive call: moves toward base case (unless ...)

Faster exponentiation

- How many recursive calls are made to compute \(2^{1024}\)?
  - How many multiplies on each call? Is this better?

```java
public static double power(double x, int n){
    if (n == 0) {
        return 1.0;
    }
    double semi = power(x, n/2);
    if (n % 2 == 0) {
        return semi*semi;
    }
    return x * semi * semi;
}
```

- What about an iterative version of this function?
### Recursive example 1

```java
double power(double x, int n) // post: returns x^n {
  if (n == 0) {
    return 1.0;
  } else {
    return x * power(x, n-1);
  }
}
```

Return value:

### Recursive example 2

```java
double fasterPower(double x, int n) // post: returns x^n {
  if (n == 0) {
    return 1.0;
  } else {
    double semi = fasterPower(x, n/2);
    if (n % 2 == 0) {
      return semi*semi;
    } else {
      return x * semi * semi;
    }
  }
}
```

Return value:

### Recursive example 3

```java
String mystery(int n) {
  if (n < 2) {
    return "" + n;
    WriteBinary(-n);
  } else {
    return mystery(n / 2) + (n % 2);
  }
}
```

Return value:

### Back to Recursion

- **Recursive functions have two key attributes**
  - There is a *base case*, sometimes called the *exit case*, which does **not** make a recursive call
    - See print reversed, exponentiation
  - All other cases make a recursive call, with some parameter or other measure that decreases or moves towards the base case
    - Ensure that sequence of calls eventually reaches the base case
    - “Measure” can be tricky, but usually it’s straightforward

- **Example: finding large files in a directory (on a hard disk)**
  - Why is this inherently recursive?
  - How is this different from exponentation?
Thinking recursively

- Problem: find the largest element in an array
  - Iteratively: loop, remember largest seen so far
  - Recursive: find largest in [1..n), then compare to 0th element

public static double max(double[] a){
    double maxSoFar = a[0];
    for(int k=1; k < a.length; k++) {
        maxSoFar = Math.max(maxSoFar, a[k]);
    }
    return maxSoFar;
}

In a recursive version what is base case, what is measure of problem size that decreases (towards base case)?

Recursive Max

public static double recMax(double[] a, int index){
    if (index == a.length-1) { // last element, done
        return a[index];
    }
    double maxAfter = recMax(a, index+1);
    return Math.max(a[index], maxAfter);
}

What is base case (conceptually)?
- Do we need variable maxAfter?
- We can use recMax to implement arrayMax as follows
  return recMax(a, 0);

Recognizing recursion:

public static void change(String[] a, int first, int last){
    if (first < last) {
        String temp = a[first]; // swap a[first], a[last]
        a[first] = a[last];
        a[last] = temp;
        change(a, first+1, last-1);
    }
}
// original call (why?): change(a, 0, a.length-1);

- What is base case? (no recursive calls)
- What happens before recursive call made?
- How is recursive call closer to the base case?

Recursive methods sometimes use extra parameters; helper methods set this up

The Power of Recursion: Brute force

- Consider the TypingJob APT problem: What is minimum number of minutes needed to type n term papers given page counts and three typists typing one page/minute? (assign papers to typists to minimize minutes to completion)
  - Example: {3, 3, 3, 5, 9, 10, 10} as page counts
  - How can we solve this in general? Suppose we’re told that there are no more than 10 papers on a given day.
    - How does this constraint help us?
    - What is complexity of using brute-force?
Recasting the problem

- Instead of writing this function, write another and call it

```java
// @return min minutes to type papers in pages
int bestTime(int[] pages) {
    return best(pages, 0, 0, 0, 0);
}
```
- What cases do we consider in function below?

```java
int best(int[] pages, int index, int t1, int t2, int t3) {
    // returns min minutes to type papers in pages
    // starting with index-th paper and given
    // minutes assigned to typists, t1, t2, t3
    //
    return best(pages, 0, 0, 0, 0);
}
```

Recursive example: BlobCount

- How do we find images? Components? Paths?
- Create information from data

Print words entered, but backwards

- Can use an ArrayList, store all the words and print in reverse order
  - Probably the best approach, recursion works too

```java
public void printReversed(Scanner s) {
    if (s.hasNext()) { // reading succeeded?
        String word = s.next(); // store word
        printReversed(s); // print rest
        System.out.println(word); // print the word
    }
}
```
- The function printReversed reads a word, prints the word only after the clones finish printing in reverse order
  - Each clone has own version of the code, own word variable
  - Who keeps track of the clones?
  - How many words are created when reading N words?
    - What about when ArrayList<String> used?

From programming techniques to Java

- Is recursion a language independent concept?
  - Do all languages support recursion?
  - What are the issues in terms of computer/compiler/runtime support?

- We use a language and its libraries, do we study them?
  - Should we know how to implement ArrayList
    - What are the building blocks, what are our tools
  - Should we know how to implement different sorts
    - Should we know how to call existing sorts
Why we study recurrences/complexity?
- Tools to analyze algorithms
- Machine-independent measuring methods
- Familiarity with good data structures/algorithms
- What is CS person: programmer, scientist, engineer?
  scientists build to learn, engineers learn to build
- Mathematics is a notation that helps in thinking, discussion, programming

Recurrences
- Summing Numbers
  ```java
  int sum(int n)
  {
    if (0 == n) return 0;
    else return n + sum(n-1);
  }
  ```
- What is complexity? justification?
  - $T(n) = \text{time to compute sum for } n$
  - $T(n) = T(n-1) + 1$
  - $T(0) = 1$
  - instead of 1, use $O(1)$ for constant time
    - independent of $n$, the measure of problem size

Solving recurrence relations
- plug, simplify, reduce, guess, verify?
  - $T(n) = T(n-1) + 1$
  - $T(0) = 1$
  - $T(n) = T(n-1) = T(n-1-1) + 1$
  - $T(n) = [T(n-2) + 1] + 1 = T(n-2)+2$
  - $T(n-2) = T(n-2-1) + 1$
  - $T(n) = [(T(n-3) + 1) + 1] + 1 = T(n-3)+3$

Now, let $k=n$, then $T(n) = T(0)+n = 1+n$
- get to base case, solve the recurrence: $O(n)$

Complexity Practice
- What is complexity of `Build?` (what does it do?)
  ```java
  ArrayList<Integer> build(int n)
  {
    if (0 == n) return new ArrayList<Integer>(); // empty
    ArrayList<Integer> list = build(n-1);
    for(int k=0;k < n; k++)
    {
      list.add(n);
    }
    return list;
  }
  ```
- Write an expression for $T(n)$ and for $T(0)$, solve.
Recognizing Recurrences

- Solve once, re-use in new contexts
  - T must be explicitly identified
  - n must be some measure of size of input/parameter
    - T(n) is the time for quicksort to run on an n-element vector

\[
T(n) = T(n/2) + O(1) \quad \text{binary search} \quad O(\log n) \\
T(n) = T(n-1) + O(1) \quad \text{sequential search} \quad O(n) \\
T(n) = 2T(n/2) + O(1) \quad \text{tree traversal} \quad O(n \log n) \\
T(n) = 2T(n/2) + O(n) \quad \text{quicksort} \quad O(n \log n) \\
T(n) = T(n-1) + O(n) \quad \text{selection sort} \quad O(n^2) \\
\]

- Remember the algorithm, re-derive complexity

Backtracking, Search, Heuristics

- Many problems require an approach similar to solving a maze
  - Certain mazes can be solved using the “right-hand” rule
  - Other mazes, e.g., with islands, require another approach
  - If you have “markers”, leave them at intersections, don’t explore the same place twice

- What happens if you try to search the web, using links on pages to explore other links, using those links to ...
  - How many web pages are there?
  - What rules to webcrawlers/webspiders follow?
    - Who enforces the rules?
  - Keep track of where you’ve been don’t go there again
  - Any problems with this approach?

Backtracking with Boggle

- Boggle\(^{\text{TM}}\) played on 4x4 board
  - Other sizes possible
  - Form words by connecting letters horizontally, vertically, diagonally
  - Cannot re-use letters (normally)

- Two approaches
  - Build words from connections, find partial words in dictionary
  - Look up every word in the dictionary on the board

- Which is better? How is backtracking used?

Classic problem: N queens

- Can queens be placed on a chess board so that no queens attack each other?
  - Easily place two queens
  - What about 8 queens?

- Make the board NxN, this is the N queens problem
  - Place one queen/column
  - # different tries/column?

- Backtracking
  - Use “current” row in a col
  - If ok, try next col
  - If fail, back-up, next row
Backtracking idea with N queens

- Try to place a queen in each column in turn
  - Try first row in column C, if ok, move onto next column
  - If solved, great, otherwise try next row in column C, place queen, move onto the next column
    - Must unplace the placed queen to keep going
- What happens when we start in a column, where to start?
  - If we fail, move back to previous column (which remembers where it is/failed)
  - When starting in a column anew, start at beginning
    - When backing up, try next location, not beginning
- Backtracking in general, record an attempt go forward
  - If going forward fails, undo the record and backup

Basic ideas in backtracking search

- We need to be able to enumerate all possible choices/moves
  - We try these choices in order, committing to a choice
  - If the choice doesn’t pan out we must undo the choice
    - This is the backtracking step, choices must be undoable
- Process is inherently recursive, so we need to know when the search finishes
  - When all columns tried in N queens
  - When all board locations tried in boggle
  - When every possible moved tried in Tic-tac-toe or chess?
    - Is there a difference between these games?
- Summary: enumerate choices, try a choice, undo a choice, this is brute force search: try everything

Computer v. Human in Games

- Computers can explore a large search space of moves quickly
  - How many moves possible in chess, for example?
- Computers cannot explore every move (why) so must use heuristics
  - Rules of thumb about position, strategy, board evaluation
  - Try a move, undo it and try another, track the best move
- What do humans do well in these games? What about computers?
  - What about at Duke?