Last Time

• Strings
  – They’re important
  – We need to sort them

• Key-Indexed Counting

• LSD
## LSD String Sort – Visual

<table>
<thead>
<tr>
<th>input (W = 7)</th>
<th>$d = 6$</th>
<th>$d = 5$</th>
<th>$d = 4$</th>
<th>$d = 3$</th>
<th>$d = 2$</th>
<th>$d = 1$</th>
<th>$d = 0$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>4PGC938</td>
<td>2IYE230</td>
<td>3CIO720</td>
<td>2IYE230</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>3ATW723</td>
<td>1ICK750</td>
<td>1ICK750</td>
</tr>
<tr>
<td>2IYE230</td>
<td>3CIO720</td>
<td>3CIO720</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>3CIO720</td>
<td>1ICK750</td>
<td>1ICK750</td>
</tr>
<tr>
<td>3CIO720</td>
<td>1ICK750</td>
<td>3ATW723</td>
<td>2RLA629</td>
<td>4PGC938</td>
<td>4PGC938</td>
<td>3CIO720</td>
<td>10HV845</td>
<td>10HV845</td>
</tr>
<tr>
<td>1ICK750</td>
<td>1ICK750</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>2IYE230</td>
<td>10HV845</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>10HV845</td>
</tr>
<tr>
<td>10HV845</td>
<td>3CIO720</td>
<td>2RLA629</td>
<td>3CIO720</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>2IYE230</td>
<td>2IYE230</td>
<td>2IYE230</td>
</tr>
<tr>
<td>4JZY524</td>
<td>3ATW723</td>
<td>2RLA629</td>
<td>3CIO720</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>2RLA629</td>
</tr>
<tr>
<td>1ICK750</td>
<td>4JZY524</td>
<td>2IYE230</td>
<td>3ATW723</td>
<td>3CIO720</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>2RLA629</td>
<td>2RLA629</td>
</tr>
<tr>
<td>3CIO720</td>
<td>10HV845</td>
<td>4PGC938</td>
<td>1ICK750</td>
<td>3CIO720</td>
<td>10HV845</td>
<td>2RLA629</td>
<td>10HV845</td>
<td>3ATW723</td>
</tr>
<tr>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>2RLA629</td>
<td>10HV845</td>
<td>3CIO720</td>
<td>3CIO720</td>
</tr>
<tr>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>2RLA629</td>
<td>10HV845</td>
<td>3ATW723</td>
<td>4PGC938</td>
<td>3CIO720</td>
</tr>
<tr>
<td>2RLA629</td>
<td>4PGC938</td>
<td>10HV845</td>
<td>10HV845</td>
<td>3ATW723</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>4JZY524</td>
<td>4JZY524</td>
</tr>
<tr>
<td>2RLA629</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>3ATW723</td>
<td>2IYE230</td>
<td>2RLA629</td>
<td>4JZY524</td>
<td>4JZY524</td>
</tr>
<tr>
<td>3ATW723</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>4PGC938</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>4PGC938</td>
<td>4PGC938</td>
<td></td>
</tr>
</tbody>
</table>
MSD String Sort

• “Most-Significant Digit First” (MSD) String Sort
  – General-purpose string sort, where strings are not necessarily all the same length.
  – Strings that start with “a” should appear before strings that start with “b,” and so forth.
  – We use key-indexed counting to sort the strings according to their first character, then recursively sort the subarrays corresponding to each first character, etc.
  – Similar to Quicksort, MSD String Sort partitions the array into subarrays that can be sorted independently.
Handling Unequal String Lengths

- Which comes first, “she” or “shell”?  
  - The subarray for strings whose characters have all been examined should come first.  
  - We do not want to recursively sort this subarray.

- We’ll handle this case by defining a custom `charAt()` function, which will return -1 if the requested character position is beyond the length of the string.
  - Because -1 is a possible return value, we now have R+1 different possible character values at each string position.
  - This means we need to add 1 to each returned value, in order to index `count[]` with a nonnegative number.
int M = 15; //cutoff to Insertion Sort
int R = 256;
String[] aux = new String[N];

void sort(String a[], int lo, int hi, int d) {
    if(hi <= lo + M) {
        Insertion.sort(a, lo, hi, d);   return;  } //if
    int count[] = new int[R+2];
    for(int i = lo; i <= hi; i++)
        count[charAt(a[i], d)+2]++;
    for(int r = 0; r < R+1; r++)
        count[r+1] += count[r];
    for(int i = lo; i <= hi; i++)
        aux[count[charAt(a[i], d)+1]++] = a[i];
    for(int i = lo; i <= hi; i++)
        a[i] = aux[i - lo]
    for(int r = 0; r < R; r++)     // recursive sort for each char val
        sort(a, lo+count[r], lo+count[r+1]-1, d+1);
} //sort
MSD String Sort – Visual

input
she sells seashells by the sea shore the she sells surely are seashells surely the seashells

output
are by by seashells sea seashells seashells seashells seashells seashells seashells

03/05/2015
MSD String Sort – Analysis

• Why include cutoff to Insertion Sort?
  – As we partition, we’ll eventually end up with a huge number of tiny subarrays, where we’ll be spending most of our sorting time.

• Worst case for MSD String Sort is when all keys are equal – we would need a recursive call for every character, when the array is already sorted to begin with.
  – Same argument goes for a large number of strings with the same prefix.
MSD String Sort – Analysis

• Running time of MSD String Sort depends on the data.
  – For random inputs, MSD examines just enough characters to distinguish among the keys. Running time is sublinear in the number of characters in the data.
  – For nonrandom inputs, MSD still could be sublinear, but might need to examine more characters than in the random case.
  – In the worst case, MSD examines all the characters in the keys, so the running time is linear in the number of characters in the data.
MSD String Sort – Analysis

- **Theorem:** To sort $N$ random strings from an $R$-character alphabet, MSD String Sort examines about $\sim N \times \log_R(N)$ characters in the average case.

- **Proof:** We expect the subarrays to all be roughly the same size, so the recurrence relation $C_N = RC_{N/R} + N$ approximately describes the performance.
MSD String Sort – Analysis

• **Theorem:** MSD String Sort uses between $8N + 3R$ and $\sim 7wN + 3wR$ array accesses to sort $N$ strings taken from an $R$-character alphabet, where $w$ is the average string length.

• **Proof:** Immediate from the code. In the best case, MSD uses just one pass. In the worst case, it performs like LSD.
# Summary

<table>
<thead>
<tr>
<th>algorithm</th>
<th>stable?</th>
<th>inplace?</th>
<th>running time</th>
<th>extra space</th>
<th>sweet spot</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort for strings</td>
<td>yes</td>
<td>yes</td>
<td>between $N$ and $N^2$</td>
<td>1</td>
<td>small arrays, arrays in order</td>
</tr>
<tr>
<td>quicksort</td>
<td>no</td>
<td>yes</td>
<td>$N \log^2 N$</td>
<td>$\log N$</td>
<td>general-purpose when space is tight</td>
</tr>
<tr>
<td>mergesort</td>
<td>yes</td>
<td>no</td>
<td>$N \log^2 N$</td>
<td>$N$</td>
<td>general-purpose stable sort</td>
</tr>
<tr>
<td>3-way quicksort</td>
<td>no</td>
<td>yes</td>
<td>between $N$ and $N \log N$</td>
<td>$\log N$</td>
<td>large numbers of equal keys</td>
</tr>
<tr>
<td>LSD string sort</td>
<td>yes</td>
<td>no</td>
<td>$NW$</td>
<td>$N$</td>
<td>short fixed-length strings</td>
</tr>
<tr>
<td>MSD string sort</td>
<td>yes</td>
<td>no</td>
<td>between $N$ and $Nw$</td>
<td>$N + WR$</td>
<td>random strings</td>
</tr>
</tbody>
</table>
String Searches

• How can we search for a string?
  – Store them in a data structure that we’ve already talked about? Sequential search, binary search, BST, 2-3 Tree, Red-Black Tree, Hash table?

• What if we had a data structure designed with complexity $W$, where $W$ is the length of the string?
  – Search hits take time proportional to the length of the search key.
  – Search misses take a few characters, until we know that the search key cannot be in the data structure.
Trie

• Composed of nodes, just like our other tree data structures.
  – One root node.
  – Each node has 1 value and $R$ child links, corresponding to the $R$ possible next letters.
Trie – Visual

characters are implicitly defined by link index

each node has an array of links and a value
Trie

- Composed of nodes, just like our other tree data structures.
  - One root node.
  - Each node has 1 value and $R$ child links, corresponding to the $R$ possible next letters.

- Search: Follow the child nodes. If you get to the end of the string and there’s a value, return the value. If not, return null. If you ever hit a null link, return null.

- Insertion: Follow the search process. If you get to the end of the string, add the value. If you ever hit a null node, add new nodes.
Trie

Value get(String key) {
    Node x = get(root, key, 0);
    if (x == null) {
        return null;
    } //if
    return (Value) x.val;
} //get

Node get(Node x, String key, int d) {
    if (x == null) {
        return null;
    } //if
    if (d == key.length()) {
        return x;
    } //if
    char c = key.charAt(d);
    return get(x.next[c], key, d+1);
} //get
Node put(Node x, String key, Value val, int d) {
    if (x == null) {
        x = new Node();
    } //if
    if (d == key.length()) {
        x.val = val;
        return x;
    } //if
    char c = key.charAt(d);
    x.next[c] = put(x.next[c], key, val, d+1);
    return x;
} //put
Trie – Simpler Visual

key: value

- she: 0
  - h
  - e

- by: 4
  - b
  - y

- sells: 1
  - e
  - l

- sea: 2
  - e
  - a
  - n

- the: 5
  - t
  - h

- sea: 6
  - a
  - n

- shells: 3
  - e
  - l
  - l

- shore: 7
  - o
  - h

Key is sequence of characters from root to value.

Nodes corresponding to characters at the end of the key do not exist, so create them and set the value of the last one.

Value is in node corresponding to last character.

One node for each key character.
Trie – Complications

• How do we keep track of the size of the trie?
  – Eager – Keep a global counter
  – Very eager – Keep a local counter, like in a BST
  – Lazy – Recurse through the whole trie counting

• How can we iterate through the trie?
  – Recursively, building strings as we go

• How can we delete?
  – Set value to null. If the node has any non-null links, we’re done. Otherwise, recursively traverse up the tree until we find a node with a non-null link.
Trie – Analysis

• **Theorem:** The number of array accesses when searching in a trie or inserting a key into a trie is at most 1 + the length of the key.

• **Proof:** Immediate from the code.
  – The `put()` and `get()` implementations carry the `d` argument.
  – `d` starts at 0, increments at each level, and is used to stop the recursion when it reaches the key length.
Trie – Analysis

• **Theorem:** The average number of nodes examined for a search miss in a trie built from \( N \) random keys over an alphabet of size \( R \) is \( \sim \log_R(N) \).

• **Proof:** ...is messy, but here’s the basic idea:
  – \( R \) gives us the number of different links at each node.
  – \( N \) gives us the “fullness” of the trie. The bigger \( N \) is, the more nodes we may hit before we have a miss.
  – As we move down the trie, our probability of a miss increases at each level.
Trie – Analysis

• **Theorem:** The number of links in a trie is between $RN$ and $RNw$, where $w$ is the average key length.

• **Proof:** Immediate from the code.
  – Every key in the trie has a node containing $R$ links, so the number of links is at least $RN$.
  – If the first characters of all of the keys are different, then there is a node with $R$ links for every key character, so the number of links is $R$ times the total number of characters, or $RNw$.

• In other words, lots of wasted space.
de la Briandais Trie

• Old version: Node = value + $R$ links
• New version: Node = value + 2 links
  – Link 1: reference to child node (move to next level)
  – Link 2: reference to sibling node (stay on same level)

• Notice that now we need to store the key as a character, rather than use them as indices.
de la Briandais Trie

bye, by, get, got, gets

```
Root
   B
  /   \
Y,1  E,0
 /     / \
∅  ∅  ∅  ∅
```

```
G
/   \\
E
/ \ \\
T,2 T,3
/ \ / \\
∅  S,4  ∅
```

```
∅
```

∅
de la Briandais Trie - Analysis

• In the worst case, with an $R$-character alphabet and a string of length $w$, our search time is $Rw$, because we search through all $R$ characters on level $i$ before moving to level $i + 1$.

• In the average case, our search time is $\sim \log_R(w)$, since we won’t search through all of the characters before moving levels, and based on the level-sparsity rationale from the regular trie analysis.
  – For small $N$, search time is just $\sim w$. 
Any Questions?