Partners are HIGHLY encouraged, but not required

Huffman and Sorting

Primitives

• How are characters stored in memory?
  • 011101000110001101110000

01101000110001101110000
ASCII

- American Standard Code for Information Interchange
  - Character encoding scheme
  - Characters mapped to numbers
    - A – 65
    - a – 97
    - ' ' (space) – 32

ASCII

<table>
<thead>
<tr>
<th>t</th>
<th>c</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>116</td>
<td>99</td>
<td>112</td>
</tr>
<tr>
<td>01110100</td>
<td>01100011</td>
<td>01110000</td>
</tr>
</tbody>
</table>

http://en.wikipedia.org/wiki/ASCII
Primitives

• How are characters stored in memory?

  • 011101000110001101110000

ASCII

• 011101000110001101110000

  Each 0 or 1 is a ‘bit’
  Each character is made of 8 bits
  8 bits = 1 byte

Frederick P. Brooks Jr.
Memory

- 8 bits = 1 byte
- 1024 bytes = 1 kb
- 1 character saved in 1 byte

497 kB (kiloBytes)*
508,928 Bytes
508,928 characters

Compression

How do they do that????
Compression

- ASCII – map each character to number represented as 8-bits

- Have to find a new mapping that uses fewer bits
  - 8 bits represent 256 characters
  - How do we represent 256 characters in fewer than 8 bits per character?

Game time
Compression

- What have we learned from Wheel of Fortune?

Huffman

- Variable length encoding
  - From our file
    - some characters are more common than others
    - encode
      - common characters < bits
      - uncommon characters > bits
Huffman

- Build a MAP

- AACCCABDE
  - A : 3
  - C : 3
  - B : 1
  - D : 1
  - E : 1

Huffman coding

AACCCABDE

- generate frequencies
- make tree for each character
- add trees to PRIORITY QUEUE
- while (more than one tree)
  - remove two smallest trees (from PRIORITY QUEUE)
  - merge trees
  - add new tree to PRIORITY QUEUE
Huffman

- Huffman (Huff) != Hough
- Hough transforms – Shape detection in images

Family Tree

BSP Trees
Pixel Planes
Tele-immersion
public void sort(int[] array) {
    if (array.length > 1) {
        int half = array.length / 2;
        int[] a1 = Arrays.copyOfRange(array, 0, half);
        int[] a2 = Arrays.copyOfRange(array, half, array.length);
        sort(a1);
        sort(a2);
        merge(array, a1, a2);
    }
}

private void merge(int[] array, int[] a1, int[] a2) {
    int len1 = a1.length; int t1 = 0;
    int len2 = a2.length; int t2 = 0;
    for (int i = 0; i < array.length; i++) {
        if (t2 == len2 || (t1 < len1 && a1[t1] < a2[t2])) {
            array[i] = a1[t1];
            t1++;
        } else {
            array[i] = a2[t2];
            t2++;
        }
    }
}
## Sorting

<table>
<thead>
<tr>
<th>Sorting Algorithm</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

## Quicksort

The Quicksort algorithm is shown in a tree structure, where the elements are sorted through a series of comparisons and partitions.
## Sorting

<table>
<thead>
<tr>
<th>Sorting Algorithm</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

## Bubble Sort

- Bubble Sort
## Sorting

### Sorting Algorithm

<table>
<thead>
<tr>
<th>Sorting Algorithm</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubblesort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Sorting

<table>
<thead>
<tr>
<th>Sorting Algorithm</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heapsort</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubblesort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertionsort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

### Bogo Sort

Start

Is Sorted?

- Yes → End
- No → Shuffle Array
Sorting

<table>
<thead>
<tr>
<th>Sorting Algorithm</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubblesort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertionsort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bogosort</td>
<td>$O(n)$</td>
<td>$n \times n!$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

INEFFECTIVE Sorts

```plaintext
DEFINE PARAMEDELIIZATION_MERGE_SORT(LIST):
  IF LENGTH(LIST) < 2:
    RETURN LIST
  PHASE = INT(LENGTH(LIST) / 2)
  A = PARAMEDELIIZATION_MERGE_SORT(LIST[0:PHASE])
  B = PARAMEDELIIZATION_MERGE_SORT(LIST[PHASE:])
  // UMMITY
  RETURN A + B // HERE'S THE DANGER

DEFINE FARTY_BOOGSORT(LIST):
  // AN OPTIMIZED BOOGSORT
  // BONUS IN O(n \log n)
  FOR N FROM 1 TO LOG(LENGTH(LIST)):
    SHUFFLE(LIST):
    IF BOOGSORTED(LIST):
      RETURN LIST
    ELSE:
      RETURN "KERNEL PAGE FAULT (PROB CODE 2)"

DEFINE JOHNEY_HAND_QUICKSORT(LIST):
  OR SO YOU CHOOSE A PHAT
  THEN SHUFFLE THE LIST IN HALF
  FOR EACH HALF:
    CHECK TO SEE IF IT'S SORTED
    IF NOT, IT DOESN'T MATTER
    COMPARE EACH ELEMENT TO THE PHAT
    THE SMALLER ONES GO IN A NEW LIST
    THE BIGGER ONES GO INTO UM
    THE SECOND LIST FROM BREN
    HANG ON, LIST ME NAME THE LISTS
    THIS IS LIST A
    THE NEW ONE IS LIST B
    PUT THE BIG ONES AND LIST B
    NOW MOVE THE SECOND LIST
    CALL IT LIST UM, A2
    WHEN ONE WHO THE PHAT IS?
    SCRAMBLE ALL THAT
    IT JUST RECURSIVELY CALLS ITSELF
    UNTIL BOTH LISTS ARE EMPTY
    RIGHT?
    NOT EMPTY, BUT YOU know WHAT I MEAN?
    AREN'T ALLOWED TO USE THE STANDARD LIBRARIES?

DEFINE PINGPONG_SORT(LIST):
  IF BOOGSORTED(LIST):
    RETURN LIST
  FOR N FROM 1 TO 100000:
    PHASE = RANDOM[LENGTH(LIST)]
    LIST = LIST[PHASE] + LIST[PHASE]
    IF BOOGSORTED(LIST):
      RETURN LIST
    ELSE:
      RETURN "THIS CAN'T BE HAPPENING"
  IF BOOGSORTED(LIST):
    IF BOOGSORTED(LIST):
      RETURN "COME ON, COME ON"
    RETURN LIST
  // ON EXIT
  // I'M GONNA BE IN SO MUCH TROUBLE
  LIST = [ ]
  SYSTEM("SHUTDOWN -H +S")
  SYSTEM("RAID -AF +A")
  SYSTEM("SNMP -R -F")
  SYSTEM("Sudo -S O, C, V") // PORTABILITY
  RETURN [1, 2, 3, 4, 5]
```

4/13/14
Take home

• There are a lot of sorting algorithms
• Some are better than others
• $O(n \log n)$ is good!
• Do NOT write your own sorting algorithm
  • It has been done for you