CSE 100: RUNTIME ANALYSIS: HUFFMAN AND BST
What is the best possible average length of a coded symbol with this frequency distribution? (why?)

A. 1
B. 2
C. 3
D. log(2)

Whenever we refer to the "best possible average code length" given a frequency distribution, think "Entropy": i.e., the metric that gives the best possible result, i.e. the metric that gives the best possible result. All actual codes will either do as well as the entropy or worse!

$$H (entropy) = \sum_i p_i \log_2 \frac{1}{p_i} = - \sum_i p_i \log_2 p_i$$

$$H = 0.25 \times -\log(0.25) + 0.25 \times -\log(0.25) + 0.25 \times -\log(0.25) + 0.25 \times -\log(0.25) = 2$$
Two quantities that we care about when we talk about encoding:
1. The expected (average) length of a code word. Using a given code and given a frequency distribution, what is the weighted average of bits needed to encode a symbol.
2. The entropy of the distribution: Given a frequency distribution, what is the minimum possible number of bits per symbol, on average, necessary for any code.

\[ L \text{ (length)} = \sum_{i} p_i \times \text{len}(\text{code word}_i) = 2 \]

\[ H \text{ (entropy)} = \sum_{i} p_i \log_2 \left( \frac{1}{p_i} \right) = -\sum_{i} p_i \log_2 p_i = 2 \]

An optimal code is one where the expected length is as close as possible to the entropy.
Symbol Frequency
S  0.6
P  0.2
A  0.1
M  0.1

\[ H = \sum_i p_i \log_2 \frac{1}{p_i} \]

What is the best possible average length (entropy) of a coded symbol with the above frequencies?

A. 2.15  
B. 1.57  
C. 1.01  
D. 0.86

See the computation in the next slide.
What is the best possible average length of a coded symbol with these frequencies?

\[ H = \sum p_i \log_2 \frac{1}{p_i} \]

Symbol | Frequency
---|---
S | 0.6
P | 0.2
A | 0.1
M | 0.1

Symbol | Codeword
---|---
S | 00
P | 01
A | 10
M | 11

\[
H = 0.6 \cdot -\log(0.6) + 0.2 \cdot -\log(0.2) + 0.1 \cdot -\log(0.1) + 0.1 \cdot -\log(0.1) = \\
= 0.6 \cdot \log(5/3) + 0.2\log(5) + 0.1\log(10) + 0.1\log(10) \\
= 1.57
\]
How fast is BST find algorithm?

How long does it take to find an element in the tree in terms of the tree’s height, H?

**Height of a node**: the height of a node is the number of edges on the longest path from the node to a leaf.

**Height of a tree**: the height of the root of the tree.

In the worst case, we are looking to find an element that is farthest from the root.

This requires doing $H$ constant time operations.

So, worst case time to find in any BST of height $H$ is $O(H)$.

$O(\log_2 N) \leq O(H) \leq O(N)$

*Balanced BST*  *Completely unbalanced* i.e. linked list

Next slides prove $O(H) = O(\log_2 N)$ for a balanced BST.
Relating H (height) and N (#nodes) find is $O(H)$, we want to find a $f(N) = H$

How many nodes are on level $L$ in a completely filled binary search tree?
A. 2  
B. $L$  
C. $2^L$  
D. $2^L$
Relating $H$ (height) and $N$ (#nodes) find is $O(H)$, we want to find a $f(N) = H$

$$N = \sum_{L=0}^{H} 2^L = 2^{H+1} - 1$$

Finally, what is the height (exactly) of the tree in terms of $N$?

$$H = \log_2(N + 1) - 1$$

$$O(H) \in O(\log_2(N+1) - 1) \in O(\log_2 N)$$

And since we knew finding a node was $O(H)$, we now know it is $O(\log_2 N)$
Worst case analysis

• Are binary search trees *really* faster than linked lists for finding elements?

  ✔ A. Yes, when they are balanced
  ✔ B. No, when they are very unbalanced
Heap-implemented priority queues

Think of the heap as a binary tree

Array implementation of heap

How long does it take to find the top item in a heap (priority queue)?
A. $O(1)$
B. $O(\log N)$
C. $O(N)$
D. $O(N \log N)$
E. $O(N^2)$

Root can be accessed in constant time
Heap-implemented priority queues

How long does it take to remove the top item in a heap (priority queue)?

A. \( O(1) \)
B. \( O(\log N) \)
C. \( O(N) \)
D. \( O(N\log N) \)
E. \( O(N^2) \)
Heap-implemented priority queues

How long does it take to add an item to a heap (priority queue)?
A. O(1)
B. O(logN)
C. O(N)
D. O(N*logN)
E. O(N^2)
Huffman’s algorithm

0. Determine the count of each symbol in the input message.

1. Create a forest of single-node trees containing symbols and counts for each non-zero-count symbol.

2. Loop while there is more than 1 tree in the forest:
   2a. Remove the two lowest count trees
   2b. Combine these two trees into a new tree (summing their counts).
   2c. Insert this new tree in the forest, and go to 2.

3. Return the one tree in the forest as the Huffman code tree.

How long does the whole thing take, using a heap-implemented Priority Queue?

N = number of distinct symbols
K = number of symbols in the file

\[ \text{Time complexity: } O(K) + O(N \log_2 N) \]

\[ \text{Worst case: } O(N \log_2 N) \text{ for a single insert into the priority queue heap.} \]
Huffman’s algorithm

0. Determine the count of each symbol in the input message. → Most of the time this step is the bottleneck!

1. Create a forest of single-node trees containing symbols and counts for each non-zero-count symbol.

2. Loop while there is more than 1 tree in the forest:
   2a. Remove the two lowest count trees
   2b. Combine these two trees into a new tree (summing their counts).
   2c. Insert this new tree in the forest, and go to 2.

3. Return the one tree in the forest as the Huffman code tree.

How long does the whole thing take, using a heap-implemented Priority Queue?

N = number of distinct symbols
K = number of symbols in the file
Average case analysis: BST

- Warning! There will be math 😊
- Why is it important that we do this?
  - So you have a hope of doing it yourself on a new data structure (perhaps one you invent?)
  - Mathematical analysis can be insightful!
Average case analysis of a “successful” find

Given a BST having:

- N nodes $x_1, .. x_N$, such that $\text{key}(x_i) = k_i$

How many compares to locate a key in the BST?

1. **Worst case**: search for the leaf node farthest from the root. In this example node 4.
2. **Best case**: search for element at the root. In this example node 5.
3. **Average case**: what we are interested in.
Given a BST having:

- N nodes \(x_1, \ldots, x_N\) such that \(\text{key}(x_i) = k_i\)
- Probability of searching for key \(k_i\) is \(p_i\)

What is the expected number of comparisons to find a key?

A. \(\sum_{i=1}^{N} p_i \cdot (\text{No. of comparisons to find } k_i)\) → Expected no. of compares

B. \(\sum_{i=1}^{N} p_i \cdot x_i\) → Expected value of nodes

C. \(\frac{\sum_{i=1}^{N} \text{No. of comparisons to find } k_i}{N}\) → Expected no. of compares if each key is searched for with equal likelihood
Number of compares to find key $k_i$ is related to the Depth of $x_i$ in the BST

- **Depth** of node $x_i$: No. of nodes on the path from the root to $x_i$ inclusive
- **Notation for depth of** $x_i$: 

![BST Diagram]

- $x_1$
- $x_2$
- $x_3$
Given a BST having:

- $N$ nodes $x_1, .. x_N$ such that $\text{key}(x_i) = k_i$
- Probability of searching for key $k_i$ is $p_i$

What is the expected number of comparisons to find a key?

A. $\sum_{i=1}^{N} p_i \cdot (\text{No. of comparisons to find } k_i)$

B. $\sum_{i=1}^{N} p_i \cdot x_i$

C. $(\sum_{i=1}^{N} \text{No. of comparisons to find } k_i) / N$
Probabilistic Assumption #1

- **Probabilistic Assumption #1:**
  All keys are equally likely to be searched (how realistic is this)?

- Thus $p_1 = \ldots = p_N = 1/N$ and the average number of comparisons in a successful find is:

\[
D_{avg}(N) = \sum_{i=1}^{N} p_i d(x_i) = \sum_{i=1}^{N} \frac{1}{N} d(x_i) = \frac{1}{N} \left( \sum_{i=1}^{N} d(x_i) \right)
\]

\[
\sum_{i=1}^{N} d(x_i) = \text{total node depth}
\]
Calculating total node depth

What is the total node depth of this tree?
A. 3
B. 5
C. 6
D. 9
E. None of these
Calculating total node depth

• In a complete analysis of the average cases, we need to look at all possible BSTs that can be constructed with same set of N keys
• What does the structure of the tree relate to?
How many possible ways can we insert three elements into a BST?

• Suppose $N=3$ and the keys are $(1, 2, 3)$
How many possible ways can we insert three elements into a BST?

• Suppose \( N = 3 \):
  
  \[(1,2,3); (1,3,2); (2,1,3); (2,3,1); (3,1,2); (3,2,1)\]

  6 possible trees

What is the total number of possibilities for an \( N \)-node BSTs?
A. \( N^N \)
B. \( N! \)
C. \( e^N \)
D. \( N*N \)
E. None of these
Relationship between order of insertion and structure of the BST

- Given a set of $N$ keys: The structure of a BST constructed from those keys is determined by the order the keys are inserted.

- Example: $N=3$. There are $N! = 3! = 6$ different orders of insertion of the 3 keys. Here are resulting trees:
Probabilistic assumption #2

- We may assume that each key is equally likely to be the first key inserted; each remaining key is equally likely to be the next one inserted; etc.
- This leads to **Probabilistic Assumption #2**
  
  *Any insertion order (i.e. any permutation) of the keys is equally likely when building the BST*

- This means with 3 keys, each of the following trees can occur with probability $1/6$

```
3, 1, 5
1, 3, 5
1, 5, 3
5, 1, 3
5, 3, 1
3, 5, 1
```
Average Case for successful Find: Brute Force Method

3, 1, 5
1, 3, 5
1, 5, 3
5, 1, 3
5, 3, 1
3, 5, 1