Lecture #19: Complexity and Search Trees

Fast Growth

• Consider Hackenmax (a function from a test some semesters ago):

```python
def Hackenmax(board, X, Y, N):
    if N <= 0:
        return 0
    else:
        return board(X, Y) + max(Hackenmax(board, X + 1, Y, N - 1), Hackenmax(board, X, Y + 1, N - 1))
```

• Time clearly depends on \( N \). Counting calls to \( \text{board} \), \( C(N) \), the cost of calling \( \text{Hackenmax}(\text{board}, X, Y, N) \), is

\[
C(N) = \begin{cases} 
0, & \text{if } N \leq 0 \\
1 + 2C(N-1), & \text{otherwise} 
\end{cases}
\]

• Using simple-minded expansion,

\[
C(N) = 1 + 2C(N-1) = 1 + 2(1 + 2C(N-2)) = 1 + 2 + 4 + 8 + \ldots + 2^{N-1} \in \Theta(2^N)
\]

Some Useful Properties

• We've already seen that \( \Theta(K_0 N + K_1) = \Theta(N) \) (\( K_0, K_1 \) here and elsewhere are constants).

• \( \Theta(N^k + N^{k-1}) = \Theta(N^k) \). Why?

• \( \Theta(|f(N)| + |g(N)|) = \Theta(\max(|f(N)|, |g(N)|)) \). Why?

• \( \Theta(\log_a N) = \Theta(\log_b N) \). Why? (As a result, we usually use \( \log_2 N = \lg N \) for all logarithms.)

• Tricky: why isn't \( \Theta(f(N) + g(N)) = \Theta(\max(f(N), g(N))) \)?

Searching, Again

• Consider the problem of searching a Python list \( L \) for some value:

```
L:
0 8 4 12 2 10 6 14 1 9 5 13 3 11 7 15
```

• If we search linearly (left to right), it will take 16 comparisons in the worst case—the length of \( L \).

• Suppose, however, we could divide our list in two, and somehow figure out quickly which of the two must contain our target, if it's to be found:

```
L:
0 8 4 12 2 10 6 14 1 9 5 13 3 11 7 15
```

• Now the cost of finding our target is at worst

\[
8 + \text{cost of deciding which list it must be in}
\]

• Consider the problem of searching a Python list \( L \) for some value:

```
L:
0 8 4 12 2 10 6 14 1 9 5 13 3 11 7 15
```

• With a cost of

\[
4 + 2 \times \text{cost of deciding which list it must be in}
\]

• As you can see, we are forming a tree.

Search Trees

• The preceding slides show the idea behind the search tree.

• The most common example is the binary search tree, where each decision is between two lists, and the decision criterion is whether the target is less than, greater than, or equal to a given value.

• These trees are a bit different from what we've been using, since they have the possibility of empty trees, such as the missing right link at the node containing 4.

• In more general search trees, as in this example, we don't have to divide sets of data exactly each time. Also, could have more than two branches.

Some Useful Properties

• \( \Theta(\log_a N) = \Theta(\log_b N) \). Why? (As a result, we usually use \( \log_2 N = \lg N \) for all logarithms.)

• Consider \( f(X, Y) \) a function from a test some semesters ago:

Fast Growth
Consider a problem with this structure:

```python
def tree_find(T, disc):
p = disc(T.label)
if p == -1:
    return T.label
elif T.is_leaf():
    return None
else:
    return tree_find(T.children[p], disc)
```

Assume that function `disc` takes (no more than) a constant amount of time.


**Questions**

- How long does the `tree_find` program (search a tree) take in the worst case on a binary tree (number of children ≤ 2)?
  - 1. As a function of $D$, the depth of the tree?
    \[ \Theta(D) \]
  - 2. As a function of $N$, the number of keys in the tree?
    \[ \Theta(N) \]
  - 3. As a function of $D$ if the tree is as shallow as possible for the amount of data?
    \[ \Theta(D) \]
  - 3. As a function of $N$ if the tree is as shallow as possible for the amount of data?

Kinds of Tree

- Assumme we are dealing with binary trees (number of children ≤ 2).
- We can classify as shalow (or, bushy) and stringy.

```
Maximally Deep (Stringy) Tree
```
```
Maximally Shallow (Bushy) Tree
```

Assume that function `disc` takes (no more than) a constant amount of time.

```
Consider a problem with this structure:
```

Slow Growth
Questions

• How long does the tree find program (search a tree) take in the worst case on a binary tree (number of children ≤ 2)?
  - As a function of \( D \), the depth of the tree? \( \Theta(D) \)
  - As a function of \( N \), the number of keys in the tree? \( \Theta(N) \)
  - As a function of \( D \) if the tree is as shallow as possible for the amount of data? \( \Theta(D) \)
  - As a function of \( N \) if the tree is as shallow as possible for the amount of data? \( \Theta(\log N) \)