Review

- Efficiency of iterative algorithms
  - In CSC148, we mainly focus on time efficiency
  - i.e. time complexity
  - We calculate/estimate a function denoting the number of operations (e.g. comparisons), and we focus on the dominant term:
    - discard all irrelevant coefficients as well as all non-dominant terms
  - We focus on the loops
    - The way the loop invariant is changed
    - If the loops are nested or sequential
  - We also watch the function calls

Efficiency of recursive algorithms?

Example 1: BST Contains

A divide and conquer problem:

```python
def bst_contains(node, value):
    if node is None:
        return False
    elif value < node.data:
        return bst_contains(node.left, value)
    elif value > node.data:
        return bst_contains(node.right, value)
    else:
        return True
```

Denote $T(n)$ as the number of operations for a tree with $n$ nodes
- Assume we always have the best tree:
  - i.e. the tree is (almost) balanced
  - $T(n) = T(n/2) + \epsilon$
- We will see the big O notation of this, shortly.
Example 2: Quick Sort

Another divide and conquer problem:
Qsort \((A, i, j)\)

if \((i < j)\)
\[
p := \text{partition}(A)
\]
Qsort \((A, i, p-1)\)
Qsort \((A, p+1, j)\)
end

Example 2: Quick Sort

Denote \(T(n)\) as the number of operations in \(\text{Qsort}\) for a list with \(n\) items.
Partition requires to traverse the whole list, i.e. \(n\) iterations.
Assume we have the best partition function: i.e. \(p\) is roughly at the middle of the list.
\(T(n) = 2T(n/2) + \epsilon\)
We will see the big \(\mathcal{O}\) notation of this, shortly.

Example 3: Merge Sort

Another, divide and conquer problem:
Msort \((A, i, j)\)

if \((i < j)\)
\[
S1 := \text{Msort}(A, i, (i+j)/2)
S2 := \text{Msort}(A, (i+j)/2, j)
\]
Merge\((S1, S2, i, j)\)
end

Example 3: Merge Sort

Denote \(T(n)\) as the number of operations in \(\text{Msort}\) for a list with \(n\) items.
Merge is to merge two sorted lists in one: the result will have \(n\) items. Hence, Merge requires \(n\) operations.
The list will be always halved.
\(T(n) = ...\)
We will see the big \(\mathcal{O}\) notation of this, shortly.

big \(\mathcal{O}\) of recurrence relations

It’s covered in CSC236.
For instance, via the Master Theorem.
If interested, read the following:
Let \(T\) be an increasing function that satisfies the recurrence relation
\[
T(n) = a T(n/b) + cn^d
\]
whenever \(n = b^k\), where \(k\) is a positive integer greater than \(1\), and \(c\) and \(d\) are real numbers with \(c\) positive and \(d\) nonnegative. Then
\[
T(n) \in \begin{cases} 
O(n^d) & \text{if } a < b^d, \\
O(n^d \log n) & \text{if } a = b^d, \\
O(n^d \log^k n) & \text{if } a > b^d.
\end{cases}
\]

big \(\mathcal{O}\) of recurrence relations

For now, we are going to accept the following common ones:

<table>
<thead>
<tr>
<th>Recurrence Relation</th>
<th>Time Complexity</th>
<th>Example Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T(n) = T(n/2) + O(1))</td>
<td>(T(n) \in \mathcal{O}(\log n))</td>
<td>(\text{bst_contains, Binary Search})</td>
</tr>
<tr>
<td>(T(n) = T(n-1) + O(1))</td>
<td>(T(n) \in \mathcal{O}(n))</td>
<td>(\text{Factorial})</td>
</tr>
<tr>
<td>(T(n) = 2T(n/2) + O(n))</td>
<td>(T(n) \in \mathcal{O}(n \log n))</td>
<td>(\text{Quart, Msort})</td>
</tr>
<tr>
<td>(T(n) = T(n-1) + T(n-2) + O(1))</td>
<td>(T(n) \in 2^n)</td>
<td>(\text{Recursive Fibubacci})</td>
</tr>
</tbody>
</table>
More insight to big $O$

- When we say an algorithm (or a function) $f(n)$ is in $O(g(n))$, we mean $f(n)$ is bounded (from up) by $g(n)$. In other words, $g(n)$ is an upper bound for $f(n)$.
- This means, there are positive constants $c$ and $n_0$ such that $f(n) \leq c \cdot g(n)$ for all $n > n_0$.
- Intuitively, this means that $f(n)$ grows slower than some fixed multiple of $g(n)$ as $n$ grows without bound.

Recall

- So, we can say:
  - $2^n$ is an upper bound for $n^2$.
  - $n^2$ is an upper bound for $n \log n$.
  - $n \log n$ is an upper bound for $n$.
Find $c$ and $n_0$ for each of these cases.

big $O$

If a function $f(n)$ is in $O(n)$, it’s also in $O(n \log n)$ and $O(n^2)$.

In general,

\[
O(1) \subseteq O(\log \log n) \subseteq O(\log n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq \cdots \subseteq O(n^2 \log n) \subseteq O(n^3) \subseteq \cdots \subseteq O(n^3) \subseteq O(2^n) \subseteq \cdots \subseteq O(3^n) \subseteq \cdots \subseteq O(n^n)
\]

However, when we are looking for an upper bound, we are required to find the tightest one.

\[F(n) = 5n^2 + 1000\] is in $O(n^2)$

Recall: Python lists and our liked lists

- Python list is a contiguous data structure
- Insertion is fast
- Deletion is slow
- Linked list is not a contiguous data structure
- Lookup is slow
- Insertion and deletion (when does not require lookup) is fast

<table>
<thead>
<tr>
<th></th>
<th>lookup</th>
<th>insert</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Linked Lists</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Recall: Balanced BST

- BST can be implemented by linked lists
- Yet, it has a property that makes it more efficient when it comes to lookup.

<table>
<thead>
<tr>
<th></th>
<th>lookup</th>
<th>insert</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Linked Lists</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>BST</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

- Yet, this comes at a price for insertion and deletion.
- Can we do better?
Can we do better?

- Assume a magical machine:
  - Input: a key
  - Output: its index value in a list
  - $O(1)$

- Well, this is a mapping machine:
  - A pair of (key, index)
  - The key is the value that we want to lookup or insert or delete, and the index is its location in the list

- And, it’s called a hash function
- And, the list is called a hash table

Hash Function

- A hash function first converts a key to an integer value,
- Then, compresses that value into an index

- Just as a simple example:
  - The conversion can be done by applying some functions to the binary values of the characters of the key
  - And the compression can be done by some modular operations

Example: (insertion)

- A class roster of up to 10 students:
- We want to enroll “ANA”
- Hash function:
  - Conversion component, for instance, returns 208 which is $65+78+65$
  - Compression component, for instance, returns 8 which is $208 \mod 10$
- So, we insert “ANA” at index 8 of the roster.

- Similarly, if we want to enroll “ADAM”:
- we insert it at index 5 of the roster (let’s call it the hash table).

Example: (lookup)

- We want to lookup “ANA”
- Hash function:
  - Conversion component, for instance, returns 208 which is $65+78+65$
  - Compression component, for instance, returns 8 which is $208 \mod 10$
- So, we check index 8 of the roster.

- Similarly, if we want to lookup “ADAM”:
  - we check index 5 of the roster (hash table).

Recall: performance

<table>
<thead>
<tr>
<th></th>
<th>lookup</th>
<th>insert</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Linked Lists</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>BST</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Hash Table</td>
<td>$O(1)^*$</td>
<td>$O(1)^*$</td>
<td>$O(1)^*$</td>
</tr>
</tbody>
</table>

* if there is no collision

Collision

- How collision can happen?
Collision

- What can we do when there is a collision?
  - Chaining

Collision

- What can we do when there is a collision?
  - Probing

Collision

- What can we do when there is a collision?
  - Double hashing

Last recall

<table>
<thead>
<tr>
<th></th>
<th>lookup</th>
<th>insert</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Linked Lists</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>BST</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Hash Table</td>
<td>O(1)*</td>
<td>O(1)*</td>
<td>O(1)*</td>
</tr>
</tbody>
</table>

- if there is no collision,
  - It’s almost impossible to prevent collision!