# Computer Science 51, section 5
# Big-O Notation and PageRank

## 1 Goal

The purpose of this section is to give you the tools to understand and compare the time and space requirements for different algorithms. Having a deeper understanding of these trade-offs will help you make better design choices.

We will also cover PageRank in preparation for part 2 of Moogle.

## 2 Tail Calls

In languages like ML, tail calls allow us to allocate stack space more efficiently.

Normally, space is allocated for every function that is called and is not deallocated until that function has fully evaluated. These function calls are very expensive in terms of both time and space. In the particular case where the function returns another function call, there is no need to keep the information from original function around; we've already finished using all of the information we need. Thus, if we plan ahead we can write recursive functions that use constant space and spend much less time switching between functions (shuffling frame pointers) on the stack!

Here we see two versions of "reduce": foldl, which utilizes this tail call optimization, and foldr, which does not. Which one is our usual reduce?

```
let rec foldr f u xs =
  match xs with
  | [] -> u
  | x::rest -> f x (foldr f u rest) ;;

let rec foldl f u xs =
  match xs with
  | [] -> u
  | x::rest -> foldl f (f x u) rest ;;
```

**Exercise 1.** Using the following definitions, what would expressions 1 and 2 evaluate to?

```
let nums = [1;2;3;4] ;;
let cons hd tl = hd::tl ;;
let more_nums = [[5;8;13];[2;3];[];[1;1]] ;;
```

| Expression 1 | Expression 2 |
|---|---|
| `foldr (+) 0 nums`<br>Answer: 10 | `foldl (+) 0 nums`<br>Answer: 10 |
| `foldr ( * ) 0 nums`<br>Answer: 0 | `foldl ( * ) 0 nums`<br>Answer: 0 |
| `foldr cons [] nums`<br>Answer: Not the same.<br>foldr returns a list in with its elements in the original order | `foldl cons [] nums`<br>Answer:<br>foldl reverses the list |
| `foldr (@) [] more_nums`<br>Answer: Not the same.<br>foldr flattens the list | `foldl (@) [] more_nums`<br>Answer: foldl flattens the list in reverse order |

**Exercise 2.** What property should functions have for the result to be the same for both foldr and foldl?

Answer: The first argument (the combining operation) must commute with itself. For example: $a + b = b + a$

# 3 Complexity and Big-O notation

The runtime of algorithms is often complicated and hard to express. Big-O notation provides a way for us to simplify the way we think about the complexity of algorithms and allows us to easily compare across different algorithms.

We can group algorithms with the same Big-O complexity into time-complexity classes, which are broad categorizations of the time it takes an algorithm to run. The broad time-complexity classes we're interested in are:

| Time Complexity | Big-O notation |
|---|---|
| constant | $O(1)$ |
| logarithmic | $O(\log n)$ |
| polynomial | $O(n^c)$ |
| exponential | $O(c^n)$ |

These are ordered from most time-efficient to least time-efficient. Why is this so? Imagine graphing equations of these types, and you can see which ones grow at a faster rate and eventually surpass the other equations.

**Exercise 3.** How would you express the following time complexities in Big-O notation? Which is the most efficient? Which is the least efficient? Can you think of any examples of algorithms that run in these times?

$n^3$ $2^n$ $n \log n$ $n^2 + n$ $5n^3 + 83n^2 + 1567$ $5^n$

Solutions:
$O(n^3)$ $O(2^n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(5^n)$

**Exercise 4** (Skip if time is limited)**.** To extract a *recurrence relation* from a function that describes its runtime, we often break it down according to its cases. Each case can typically be represented with its own equation, where we will map constant-time operations to constant values and function calls (recursive or otherwise) to applying mathematical functions.

These are some useful recurrence relations and their solutions. Make sure that you have at least an intuitive sense of why these solutions are correct.

| `T(n)` | Big-O . |
|---|---|
| $c$ | $O(c)$ |
| $c + \texttt{T(n - 1)}$ | $O(n)$ |
| $kn + \texttt{T(n - 1)}$ | $O(n^2)$ |
| $c + \texttt{T(n/2)}$ | $O(\log n)$ |
| $kn + \texttt{T(n/2)}$ | $O(n)$ |
| $kn + 2\texttt{T(n/2)}$ | $O(n \log n)$ |

**Exercise 5.** For the following functions:

- What is the asymptotic running time?
- Write the recurrence relation for its running time.
- Does the function utilize tail-calls? If not, could we easily rewrite it so it does?

1. `let split lst = foldl (fun x (a,b) -> (x::b,a)) ([],[]) lst`
   - $\texttt{T(n)} = \texttt{c} + \texttt{T(n} - 1\texttt{)}$
   - $O(n)$
   - Yes!
2. Remember `merge` from problem set 1? It takes two ordered lists and combines them into one ordered list. Its runtime is $O(n+m)$ where $n$ and $m$ are the length of the two lists being merged.

   ```
   let rec mergesort (lst:int list) =
     match lst with
       | x::y::rest -> let (a,b) = split lst in merge (mergesort a) (mergesort b)
       | _ -> lst
   ```

   - $\texttt{T(n)} = \texttt{kn} + 2 * \texttt{T(n/2)}$
   - $O(n \log n)$
   - No. No
3. Remember `partition` from problem set 1? It takes one list and splits it into two. The first list contains only elements smaller than some numbmer $x$, and the second list contains only elements greater than or equal to $x$. Its runtime is $O(n)$ in the length of the list.

   ```
   let rec quicksort (lst:int list) =
     match lst with
       | [] -> []
       | x::rest -> let (lows,highs) = partition x rest in
             quicksort lows @ (x :: quicksort highs)
   ```

   - $\texttt{T(n)} = \texttt{kn} + \texttt{T(n} - 1 - \texttt{x)} + \texttt{T(x)} + \texttt{c}$ for some $x < n$
   - $O(n^2)$. (conservative bound; usually quicker; can develop some intuition by looking at extreme cases:
     - $\texttt{T(n)} = \texttt{kn} + \texttt{c} + 2\texttt{T(n/2)}$ (always a good split)
     - $\texttt{T(n)} = \texttt{kn} + \texttt{c} + \texttt{T(n} - 1\texttt{)}$ (always a bad split))
   - No. No.

# 4 PageRank

The key insight is that the links to a page can indicate the importance of that page. Furthermore, links from important pages are better indicators than links from less important pages. This algorithm captures all of this information.

PageRank can be thought of as a model of user behavior. We assume there is a "random surfer" who is given a web page at random and keeps clicking on links, never hitting "back" but eventually gets bored and starts on another random page. The probability that the random surfer visits a page is its PageRank.

To clarify, each page is associated with the probability that it will be accessed by the random surfer. This can serve as a measure of the page's relative importance because the random surfer would be more likely to stumble upon a page that has lots of links from other important sites.

Overview of the algorithm:

1. Every page starts with equal probability of being reached. At this point we have not looked at any links! If there are ten pages on the internet then each one will start with the initial probability of $\frac{1}{10}$.
2. The random surfer follows a link with probability $(1-\alpha)$ and jumps to a random new page with probability $\alpha$. Thus, the probability of arriving at a particular page is: $\alpha \cdot P(\text{arrive randomly}) + (1-\alpha) \cdot P(\text{arrive through a link})$
3. To handle the case of nodes that have no outgoing edges, assume that each node has an implicit edge to itself.

The iterative algorithm in equations:

$$P(p_i; 0) = \frac{1}{N} \tag{1}$$

$$P(p_i; t+1) = \frac{\alpha}{N} + (1-\alpha) \sum_{p_j \in M(p_i)} \frac{P(p_j; t)}{L(p_j)} \tag{2}$$

Keep repeating step 2 until all of the probabilities converge to something close to their true value. It shouldn't take too many iterations for the probabilities to become extremely stable.

Definition of terms:

- $p_i$ and $p_j$ are pages on the internet.
- $P(p_i; t)$ is the probibility of stumbling upon page $p_i$ after $t$ iterations of this algorithm.
- $N$ is the number of pages on the internet (i.e. the number of pages you have crawled).
- $L(p_j)$ is the number of outgoing links from page $p_j$, including implicit link to itself.

- $M(p_i)$ is the set of pages that point to page $p_i$.
- $\alpha$ is the damping factor that makes this algorithm converge. It is generally agreed upon that $\alpha$ should be around 0.15.