Orders of Growth

We have seen more than one algorithm that performed the same task. How do we choose which one is best? We could just time the two algorithm but there are some weaknesses with this approach. What input do we choose? The algorithms might perform faster or slower depending on the input. What if one algorithm is faster on one input and the other is algorithm is faster on another input?

Instead, computer scientists use an *asymptotic* approach in which we look at which algorithm's time increases more rapidly as the problem size increase. To get a feel for this approach we will try out two algorithms for a sorting a deck of card. You can take index cards with numbers written on them and sort them using two different algorithms. You will try each algorithm for decks of size 4, 8, 16, and 32 and time your results. Here are the two algorithms:

Selection Sort:
You will use three positions for stacks of cards: a source stack, destination stack, and a discard stack.

Initially you should put all the cards face down on the source stack with the other two positions empty. Perform the following steps:

1. Take the top card off the source stack and put it face-up on the destination stack.
2. If the source stack is now empty then you are done. The destination stack is in numerical sorted order.
3. Otherwise, do the following steps repeatedly until the source stack is empty:
   (a) Take the card off the source stack and compare it with the top of the destination stack.
   (b) If the source card has a larger number,
      i. Take the card on the top of the destination stack and put it face down on the dicard stack.
      ii. Put the card you took from the source stack face up on the destination stack.
   (c) Otherwise put the card from the source stack face down on the discard stack.
4. Slide the discard stack over into the source position and start again with step 1.

Merge Sort:
Lay out the cards face down in a long row. We will consider these cards to initially be source ”stacks” with only one card per stack. The merge sort works by progressively merging pairs of stack so there will be fewer and fewer stacks but each stack is larger. At the end ther will be a single stack of sorted cards.

Repeat the following steps until there is a single stack of cards:

1. Merge the first tow face-down stacks using the Merge algorithm given below.
2. As long as there are at least two face-down stacks, repeat the merging with the next two stacks.
3. Flip each face-up stack over.

Merging:
You have two sorted stacks of cards to merge side by side, face down. You will be producing the result stack above the other two, face up.

Take the top card off of each source stack – one in your left hand and one in your right hand. Now do the following repeatedly, until all the cards are on the destination stack:

1. Compare the two cards you are holding.
2. Place the one with the larger number onto the destination stack, face-up.
3. With the hand you just emptied, pick up the next card from the corresponding source stack and go back to step 1. If there is no next card in the empty hand's stack because that stack is empty, put down the other card you are holding on the destination stack face-up and continue flipping the rest of the cards over onto the destination stack.

When you time our trials, you should see the growth rate on one of your sorts is much greater than the other. We should be able to determine the rate of growth without actually timing the sorts by carefully examining each of the algorithms. We will consider the number of ”steps” it takes for each of the algorithms when sorting n cards.

In Selection Sort we will consider a *pass* to be one time through steps 1 through 4 and will examine how many cards are handled on each pass.

Pass Number	Number of handled cards
pass 1	all n cards handled once or twice
pass 2	$n-1$ cards handled once or twice
pass 3	$n-2$ cards handled once or twice
...	
pass n	1 card handled

So the number of cards handled is $1 + 2 + 3 + ... + n$. How big is this sum? Well, we can check to see that the sum is less than n^2. This is because each number is no bigger than n so the entire sum is no bigger than $n + n + n + ...n = n^2$. We can also check that the sum is at least $n^2/4$, because there are $n/2$ numbers which are larger than $n/2$. Therefore the sum is somewhere between $n^2/4$ and n^2, both multiples of n^2. From this we can determine that the growth rate will be the shape of the graph n^2 and we write this as $O(n^2)$.

We will do the same analysis for Merge Sort.

Pass Number	Number of handled cards
pass 1	all n cards handled to merge n stacks into $n/2$ stacks
pass 2	all n cards handled to merge $n/2$ stacks into $n/4$ stacks
pass 3	all n cards handled to merge $n/4$ stacks into $n/8$ stacks
...	
pass ?	all n cards handled to merge 2 stacks into 1 stack

How many passes will we need? Another way to answer this is to go in the opposite direction and ask how many times time 1 need to double to reach n? Answer — $log_2 n$. Therefore the order of growth for merge sort is $O(n \; log \; n)$. This growth rate is faster than $O(n)$ but slower than $O(n^2)$. This should be confirmed by the results of the time trials.

A Haskell example

We will consider three different algorithms for computing b^n: a linear recursive algorithm, a tail recursive algorithm, and an algorithm using a stronger form of recursion.

Linear Recursion:

```
power :: Int -> Int -> Int
power b n = if n==0
                then 1
                else b * (power b (n-1))
```

Both the time and space (memory use) is proportional to the depth of the recursion (number of winding steps), which is n. Therefore both time and space is $O(n)$.

Tail Recursion:

```
powerTail :: Int -> Int -> Int
powerTail b n = pTail n 1 where
       pTail n result =
             if n==0
             then result
             else pTail (n-1) (result * b)
```

The time is proportional to the depth of the recursion, which is n. Therefore time complexity is $O(n)$. The space complexity, however will always be constant regardless of the values of the inputs. Therefore the space complexity is $O(1)$.

Strong Recursion:
For this version, observe that if n is even then $b^n = (b^{n/2})^2$.

```
power2 :: Int -> Int -> Int
power2 b n = if n==0
                then 1
                else if even n
                then square (power2 b (n ‘div‘ 2))
                else b * (power2 b (n-1))
```

The time is proportional to the depth of the recursion. Just as in Merge Sort, if n is a power of 2, it will take $log\ n$ steps to get to the base case. Therefore the time and space compexities are both $O(log\ n)$.