Chapter 1

Big-Oh From Code

- We grow an array by increasing its length by 1 each time.

  ```
  double[] numbers = new double[1];
  for ( int i = 0 ; i < n ; i++ ) {
    if ( i >= numbers.length ) {
      numbers = Arrays.copyOf(numbers,numbers.length+1);
    }
    numbers[i] = Math.random();
  }
  ```

  Outside (before) the loop is just simple operations, so that contributes $\Theta(1)$.

  For the loop, observe that everything in the loop body is $\Theta(1)$ except `Arrays.copyOf()`, which we expect to take time proportional of the number of elements copied i.e. $\Theta($`numbers.length`$)$. The total amount of time taken for the loop is the sum of the time taken by each iteration. Step through the code: on the first iteration $i = 0$, `numbers.length` $= 1$, and the `if` condition is false so nothing is copied and `numbers.length` doesn't change. On the next iteration $i = 1$, `numbers.length` $= 1$, and the `if` condition is true so `numbers` is copied and its length increases by 1. And so forth:

i	0	1	2	3	4	5	...	$n-1$
`numbers.length`	1	1	2	3	4	5	...	$n-1$
work to copy	0	1	2	3	4	5	...	$n-1$
other work	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	...	$\Theta(1)$

  The total time taken is the sum of the "work to copy" and "other work" entries: $\sum_{j=0}^{n-1} j + n\Theta(1)$.

  Using the sums table gives $\Theta(n^2)$ for the sum, which is faster-growing than n, so the overall running time is $\Theta(n^2)$.

- We grow an array by doubling its length each time.

```
double[] numbers = new double[1];
for ( int i = 0 ; i < n ; i++ ) {
  if ( i >= numbers.length ) {
    numbers = Arrays.copyOf(numbers,2*numbers.length);
  }
  numbers[i] = Math.random();
}
```

Outside (before) the loop is just simple operations, so that contributes $\Theta(1)$.

For the loop, observe that everything in the loop body is $\Theta(1)$ except `Arrays.copyOf()`, which we expect to take time proportional of the number of elements copied i.e. $\Theta($`numbers.length`$)$. The total amount of time taken for the loop is the sum of the time taken by each iteration. Step through the code: on the first iteration $i = 0$, `numbers.length` $= 1$, and the `if` condition is false so nothing is copied and `numbers.length` doesn't change. On the next iteration $i = 1$, `numbers.length` $= 1$, and the `if` condition is true so `numbers` is copied and its length is doubled. And so forth:

i	0	1	2	3	4	5	6	7	8	9	...	$n-1$
`numbers.length`	1	1	2	4	4	8	8	8	8	16	...	
work to copy	0	1	2	0	4	0	0	0	8	0	...	
other work	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	...	$\Theta(1)$

The total time taken is the sum of the "work to copy" and "other work" entries. For "work to copy", observe that it is a sum of powers of 2: $\sum_{j=0} 2^j$. But what's the upper limit for the sum? Assume n is a power of 2, so the last time the array grows and is copied is when $2^j = n/2$. Solving for j yields $j = logn - 1$.

Thus, the total time taken is $\sum_{j=0}^{logn-1} 2^j + n\Theta(1)$. This is a "geometric increase" sum, so using the sums table yields $\Theta(2^{logn-1}) + n\Theta(1)$. 2^{logn-1} simplifies to $n/2$, so the total time is $\Theta(n)$.

[This means that over the time it takes to insert n elements, doubling the array results in only $O(n)$ additional work in total — while the worst case behavior of a single insert is $O(n)$, when the growing time is spread over a series of n operations (a process called *amortized analysis*) each insert is effectively $O(1)$.)

- ```
  void hanoi ( int n, int src, int dst, int spare ) {
    if ( n == 1 ) {
      System.out.println(\move disk from \+src+" to \+dst);
    } else {
      hanoi(n-1,src,spare,dst);
      System.out.println(\move disk from \+src+" to \+dst);
      hanoi(n-1,spare,dst,src);
    }
  }
  ```

  Let $T(n)$ be the time for `hanoi(n,...)`. Then

  $$T(1) = \Theta(1)$$

  For the recursive case, the time taken is the time for two `hanoi(n-1,...)` calls plus $\Theta(1)$ additional time — the only non-simple steps in the body of `hanoi` are the recursive calls. This means

  $$T(n) = 2T(n-1) + \Theta(1)$$

  Using the recurrence relations table gives $\Theta(a^{n/b}) = \Theta(2^n)$.

- Mergesort.

  ```
  void mergesort ( int[] arr, int left, int right ) {
    if ( right > left ) {
      int middle = (left+right)/2;
      mergesort(arr,left,middle);
      mergesort(arr,middle+1,right);
      merge(arr,left,middle,right);
    }
  }

  void merge ( int[] arr, int left, int middle, int right ) {
    int[] merged = new int[right-left+1];
    int int i = left, j = middle+1, k = 0;
    for ( ; i <= middle && j <= right ; k++ ) {
      if ( arr[i] < arr[j] ) { merged[k] = arr[i]; i++; }
      else { merged[k] = arr[j]; j++; }
    }
    for ( ; i <= middle ; i++, k++ ) {
      merged[k] = arr[i];
    }
    for ( ; j <= right ; j++, k++ ) {
      merged[k] = arr[i];
    }
    System.arraycopy(merged,0,arr,left,merged.length);
  }
  ```

  For `mergesort`, the base case is $\Theta(1)$ (only the `if` condition is checked). For the recursive case

  $$T(n) = 2\,T(n/2) + \Theta(n)$$

  where $n = \texttt{right} - \texttt{left} + 1$. (`right` and `left` denote the range of `arr` being sorted.) For `merge`, observe that every loop iteration increments either `i` or `j` and that `i` counts from `left` to `middle` (inclusive) and `j` counts from `middle+1` to `right` (inclusive) — thus the total work for the three loops is $\Theta(n)$. `System.arraycopy` is also $\Theta(n)$ making `merge` $\Theta(n)$ overall.

  Using the recurrence relations table gives $T(n) = \Theta(n \log n)$.
  ```