```
public int delMax() {
		if (n == 0) {
			return 0;

		}

		int max = pq[1];

		exch(1, n--);
		sink(1);
		assert pq[n + 1] == max;
		pq[n + 1] = -1;
		return max;

	}
private void sink(int k) {

		int child;
		int temp = pq[k];
		while (getChild(k, 1) < n) {
			child = maxChild(k);
			if (pq[child] < temp) {
				pq[k] = pq[child];
			} else {
				break;
			}
			k = child;
		}
		pq[k] = temp;
	}
private int getChild(int child, int parent) {
		int one = parent - 1;
		int two = child + 1;
		int three = d * one;
		int four = three + two;

		return four;
	}
```

DelMax on its own does nothing but assign/swap values in the queue. However, it calls sink, which in turn calls getChild, which are more involved processes. Sink involves iterating through the entire queue, from the first child on, to find the maximum value. Since get child merely finds the value of the child at a given index, it is sink that will bottleneck the function. Since it involves iterating through the queue one time, we have O(N).

```
private void sink(int[] a, int k, int n) {

		int child;
		int temp = a[k];
		while (getChild(k, 1) < n) {
			child = maxChild(k);
			if (a[child] < temp) {
				a[k] = a[child];
			} else {
				break;
			}
			k = child;
		}
		a[k] = temp;

	}
private int getChild(int child, int parent) {
		int one = parent - 1;
		int two = child + 1;
		int three = d * one;
		int four = three + two;

		return four;
	}




public void sort(int a[]) {
		int n = a.length;
		for (int k = n / 2; k >= 1; k--) {
			sink(a, k, n);
		}
		while (n > 1) {
			exch(a, 1, n--);
			sink(a, 1, n);
		}

	}
public int[] daryHeapsort() {
		int a[] = new int[n];
		for (int i = 0; i < n; i++) {
			a[i] = pq[i];
```

```
        }
        sort(a);
        return a;
    }
```

Heapsort beings by iterating through the entire queue and copying it to an array. This makes us a start with a time of at least N. It then calls sort, which loops through the entire queue starting at the halfway point and working its way down, calling sink each time. Sink requires looping through the entire queue, so we have (N/2)*N. It then loops through the whole queue decrementing each time, causing another N*N. So we have the sum of these as our total time complexity, or $O(N+(N^2/2)+N^2)$, which is $O((3N^2/2)+N)$.