delMax time complexity

```
int delMax()
	{
		int max = arr[0];

		arr[0] = arr[currentSize - 1];

		currentSize = currentSize - 1;


		sink(0);

		return max;
	}
```

The function shown above showcases above shows the code for the delMax function. All of the code in yellow runs for constant time. The only line of code that does not run for constant time is the call to the function sink. This means that the worst case run time of this function is the worst-case run time of sink plus some constants. Based of this information the big O runtime of this function is O(worst-case run time of sink).

```
void sink(int index)
	{
		int *child;

		child = new int[numKids + 1];

		while (1)
		{
			for (int i = 1; i <= numKids; i++){
				if ((numKids*index + i) < len) {
					child[i] = (numKids*index + i);
				}
				else {

					child[i] = -1;
				}
			}


			int MaxChild = -1;
			int MaxChildLoca;

			for (int i = 1; i <= numKids; i++)
			{
				if (child[i] != -1 &&
					arr[child[i]] > MaxChild)
				{
					MaxChildLoca = child[i];
					MaxChild = arr[child[i]];
				}
			}

			if (MaxChild == -1)
				break;
```

```
                if (arr[index] < arr[MaxChildLoca])
                        swap(arr[index], arr[MaxChildLoca]);

                index = MaxChildLoca;
        }
    }
```

The code shown above showcases the sink function. The major time constraints are the while loop shown in cyan and the for loops shown in green. The while loop ends when the node that needs to be sunk is at the correct position so that it satisfies the max heap rule. In every instance that this function is called by delMax. The node that needs to be sunk is the first node at index 0. In the worst-case situation, the node will have to be sunk to the bottom level of tree. This means that the while loop will have to run O(height of tree) times. The maximum height of a heap with k children is equal to O($log_k(n)$) where n is equal to the number of nodes in the heap. This indicates that the while loop will run in the worst case O($log_k(n)$) times. Both for loops inside the loop run k times where k = maximum number of children per node. Everything else in the while loop runs for a constant amount of time. This information indicates that the run time of the sink functions in the worst case is equal to O($log_k(n)$) $*$ O(2k) = O($2k * log_k(n)$) = O($k * log_k(n)$). If the worst case run time of the sink function is O($k * log_k(n)$), then the worst case run time of delMax is also O($k * log_k(n)$).

daryHeapsort time complexity

```
int* daryHeapsort()
        {

                int *clone;
                clone = new int[currentSize - 1];
                for (int i = 0; i <= currentSize - 1; ++i) {
                        clone[i]=arr[i];
                }
                for (int i = currentSize - 1; i >= 0; i--)
                {
                        swap(clone[0], clone[i]);

                        sinkSort(clone, i);
                }
                return clone;
        }
```

The code shown above showcases the sinkfunction. The only portion of the code that does not run for a constant amount of time.is the two for loops highlighted in green. The runtime for each for loop is equal to n where n is the number of nodes in the heap the. The contents inside the first for loop run for constant amount of time. The sinkSort() function call inside the second for loop has the same worst case run time as the sink function discussed in the previous section. This means that the worst case run time of sinkSort is O($k * log_k(n)$). Based on this information, the total runtime of daryHeapsort in the worst case is O($n + n(k * log_k(n))$ = O($n(k * log_k(n))$.