Implementing Sorting Algorithms Recursively

Discussion

Any algorithm that can be written iteratively can also be written recursively, including any of the sorting algorithms. One advantage to rewriting algorithms recursively is that it can simplify correctness proofs. Also when recursion is used, we can apply some of the techniques used with recursive algorithms for determining algorithm efficiency. There may, however also be some disadvantages. In particular, when written iteratively most sorting algorithm run in constant memory, but that will never be the case for their recursive counterparts.

Shown below is the code for the selection sort consisting of two recursive methods that replace the two nested loops that would be used in its iterative counterpart.

```
void selectionSort(int array[])
{
  sort(array, 0);
}

void sort(int[] array, int i)
{
  if (i < array.length - 1)
  {
    int j = smallest(array, i);
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
    sort(array, i + 1);
  }
}

int smallest(int[] array, int j)
  {
  if (j == array.length - 1)
    return array.length - 1;
  int k = smallest(array, j + 1);
  return array[j] < array[k] ? j : k;
}
```

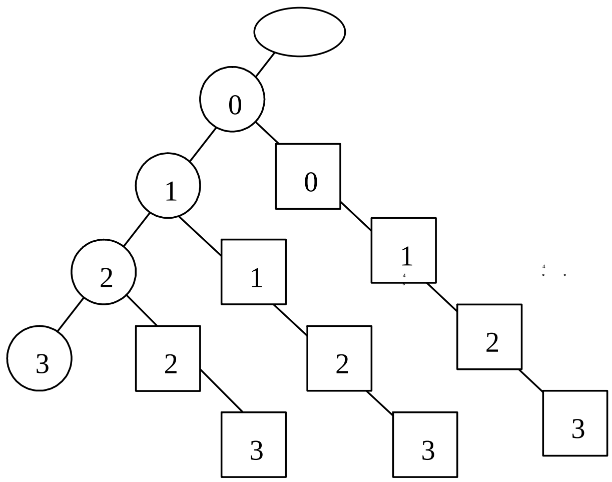
Notice that the first method is a helper method used to initialize the second parameter of the `sort` method.

Sample Problem

Draw the recursion tree for `selectionSort` when it is called for an array of length 4 with data that represents the worst case. Show the activations of `selectionSort`, `sort` and `smallest` in the tree. Explain how the recursion tree would be different in the best case.

Solution

Shown below is the recursion tree, where the oval represents the call to `selectionSort`, the circles represent calls to `sort` and the squares represents calls to `smallest`. The value in the `sort` nodes represents the parameter `i` and the value in the `smallest` nodes represents the parameter `j`.



There is no difference between the best and worst cases with the selection sort so the recursion tree for the best case would be identical.

Analyzing Recursive Sorting Algorithms

Discussion

Although not all recursive implementations of sorting algorithms contain recursive methods that execute in constant time, that is the case with the implementation of the selection sort presented above. Consequently we can make use of the fact that for such algorithms the execution time is proportional to the number of nodes in the recursion tree and the memory usage is proportional to the height of the tree.

Sample Problem

Determine a formula that counts the numbers of nodes in the recursion tree. What is Big-Θ for execution time? Determine a formula that expresses the height of the tree. What is the Big-Θ for memory?

Solution

By examining the tree from the previous problem where *n* is 4, we see that the number of nodes in the 3 branches where `smallest` is called contain 3, 4 and 5 nodes. We can express this by the summation $\sum_{i=3}^{n+1} i$. In addition there is one node above and one below, so

$t = (\sum_{i=3}^{n+1} i) + 2 = (\sum_{i=1}^{n+1} i) - 1 = \frac{(n+1)(n+2)}{2} - 1 = \frac{n^2+3n+2}{2} - 1 = \frac{n^2+3n}{2}$

We can conclude that the execution time efficiency is $\Theta(n^2)$ because $t \in \Theta(n^2)$.

From the diagram, it is easy to see that the height of the tree is the length of the right-most branch, so $h(n) = n + 2$. Consequently the efficiency of memory utilization is $\Theta(n)$ because $h \in \Theta(n)$.

Implementing the Heapsort with a Priority Queue

Discussion

Because the Java priority queue is implemented using a heap, it is possible to implement the heapsort using the Java priority queue by performing a series of `add` operations followed by a series of `remove` operations. The code to accomplish this task is shown below:

```
void sort(int[] array)
{
  PriorityQueue<Integer> queue = new PriorityQueue();
  for (int i = 0; i < array.length; i++)
    queue.add(array[i]);
  for (int i = 0; i < array.length; i++)
    array[i] = queue.remove();
}
```

Although using a heap to implement a priority queue is the most desirable implementation, the above code would work with any implementation of a priority queue.

Sample Problem

Provide a generic Java class that implements a priority queue using an unsorted list implemented with the Java `ArrayList` class. Make the implementation as efficient as possible.

Solution

```
class UnsortedPriorityQueue<T extends Comparable>
{
  private ArrayList<T> queue = new ArrayList();

  public void add(T element)
  {
    queue.add(element);
  }
  public T remove()
  {
    int largestIndex = 0;
    for (int i = 1; i < queue.size(); i++)
      if (queue.get(largestIndex).compareTo(queue.get(i)) < 0)
        largestIndex = i;
    T largestValue = queue.get(largestIndex);
    if (largestIndex == queue.size() - 1)
      queue.remove(largestIndex);
```

```
    else
      queue.set(largestIndex, queue.remove(queue.size() - 1));
    return largestValue;
  }
}
```

Analyzing the Efficiency of Algorithms that Use the Java Collection Classes

Discussion

When we use predefined classes to implement any algorithm, we must consider the efficiency of the methods in those classes when determining the efficiency of the code that we have written.

Let us consider the implementation of the heap sort implemented with the Java priority queue discussed above. Because both the `add` and `remove` methods are $\Theta(\log n)$ in the worst case, each of the loops is $\Theta(n \log n)$, so the overall method is $\Theta(n \log n)$ also. So this implementation is as efficient as the heap sort that makes direct use of a heap.

Sample Problem

Consider the sorting algorithm that use the priority queue implemented in the previous problem. Analyze its execution time efficiency in the worst case. In your analysis you may ignore the possibility that the array list may overflow and need to be copied to a larger array. Indicate whether this implementation is more or less efficient than the one that uses the Java priority queue.

Solution

The `add` method of `UnsortedPriorityQueue` consists of a single call to the `add` method of the `ArrayList` class. The latter `add` method appends to the end of the list, so it is $\Theta(1)$, which makes the `add` method of `UnsortedPriorityQueue` $\Theta(1)$, also. The first loop of the `sort` method calls `add` *n* times, so that first loop is $\Theta(n)$.

The `remove` method of `UnsortedPriorityQueue` contains a loop that executes *n* times. It calls `get` several times, which is $\Theta(1)$, so that loop is $\Theta(n)$. Following the loop is a single call to `get`, which is $\Theta(1)$. Next is a statement containing three calls that are nested. Both `set` and `size` are $\Theta(1)$. The third method, `remove`, is $\Theta(n)$ in general, but because we are always removing the last element it is $\Theta(1)$ in this case, so the whole statement is $\Theta(1)$. Consequently the loop predominates and the efficiency of the whole `remove` method of `UnsortedPriorityQueue` is $\Theta(n)$. Finally we consider the efficiency of the second loop of the `sort` method. That loop executes *n* calls to `remove`, making that loop $\Theta(n^2)$.

Because the second loop is $\Theta(n^2)$, it predominates making the overall efficiency of `sort` $\Theta(n^2)$. Consequently this implementation is less efficient than the one that uses the Java priority queue.