Lecture Notes for Advanced Algorithms

Prof. Bernard Moret

September 29, 2011

Notes prepared by Blanc, Eberle, and Jonnalagedda.

1 Average Case Analysis

1.1 Reminders on quicksort and tree sort

We start with a few reminders on quicksort and tree sort. Quicksort and tree sort are both sort algorithms, which work in a very similar way. The main difference being that tree sort usually works recursively using binary search trees (Figure 1) while quicksort uses an array. The use of arrays, being a very efficient data structure, would probably make quicksort faster in practice, but as far as the algorithmic complexity goes, they are really the same algorithms.

Algorithm 1 Tree sort

for $i = 1 \rightarrow n$ **do**
  insert $A[i]$ into tree
end for
inorder traversal

Algorithm 1 describes (at a very high level) the tree sort algorithm. The idea is pretty straightforward: Build a binary search tree while going through

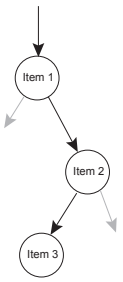

Figure 1: Binary tree

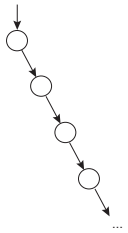

Figure 2: Degenerate binary tree

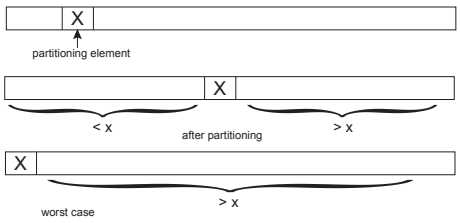

Figure 3: Quicksort

the input array and then do an inorder traversal of the tree to output the sorted array.

The inorder traversal is $\Theta(n)$ since you need to visit every elements of the original array of size n. The body of the loop is executed $\Theta(n)$ times, each of those consisting in an insert operation on a tree. This typically only takes $\Theta(\log n)$ on a well balanced binary tree, but in the case of a degenerate tree (Figure 2), this is a $\Theta(n)$ operation.

This means that this algorithm has a worst case running time of $\Theta\left(n^2\right)$. But in fact, as we will see, this algorithm runs in average time of $\Theta(n \cdot \log n)$.

Algorithm 2 Quicksort

Pick a partitioning element
Partition the array
recurse left
recurse right

Algorithm 2 gives a high-level description of quicksort. The idea is to pick an element, then place all smaller elements on its left and all bigger elements on its right. Finally we apply the sort recursively on the left sub-array and the right sub-array. Figure 3 illustrates the successive steps to perform.

Again, in some degenerate cases (an already sorted array, with the pivot being the first element of the array), we end up with a running time of $\Theta\left(n^2\right)$,

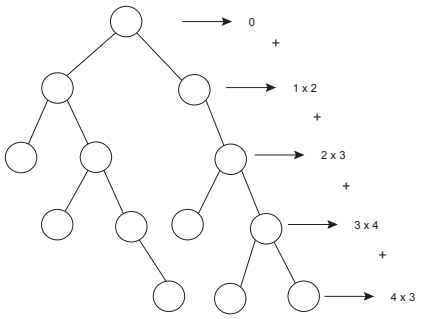

Figure 4: Cost $I(T)$ of tree

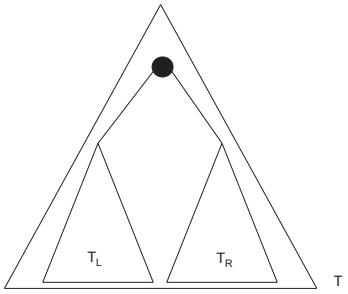

Figure 5: Subtrees

while in average we have a running time of $\Theta(n \cdot \log n)$.

1.2 Average case analysis of tree sort

The idea behind average case analysis is to think about what happens *on average* when an algorithm runs over a large number of inputs. However it is in general quite hard to get an idea of what a real world distribution of the data can be. For the treesort case, it is very hard to get a distribution of the input array. Therefore we make clever assumptions on the distribution of the data. In this case, we assume that all $n!$ orderings of an input are equally likely.

Analysing the treesort algorithm amounts to calculating the cost $I(t)$ of creating a tree:

$$I(T) = \sum_{\text{nodes}} (\text{length of root to node path})$$

We can describe I using a recurrence relation:

$$I(T) = |T| - 1 + I(T_L) + I(T_R)$$

where T_L denotes the left subtree and T_R denotes the right subtree, as shown in Figure 5.

The *key* observation here is that because all trees are equally probable (our assumption on the distribution), *any* node can be the root node. This is why we can do an average case analysis by computing the expected path length for a tree with n nodes; we write $\bar{I}(n)$, where n is the numbers of node. Here is a recursive definition of $\bar{I}$:

$$\bar{I}(n) = \frac{1}{n} \cdot \sum_{i=0}^{n-1} [n - 1 + \bar{I}(i) + \bar{I}(n - i - 1)]$$

This recurrence equation is called a full-history recurrence (it depends on every previous state). Obviously it is not of finite order, so we cannot apply our previous method. However we can apply other tricks. In particular we can do telescoping in which we subtract from the relation the same relation shifted by one:

$$\begin{aligned} n \cdot \bar{I}(n) - (n-1) \cdot \bar{I}(n-1) &= \textstyle\sum_{i=0}^{n-1} [n - 1 + \bar{I}(i) + \bar{I}(n - i - 1)] - \sum_{i=0}^{n-2} [(n - \\ &\quad 1) - 1 + \bar{I}(i) + \bar{I}((n-1) - i - 1)] \\ &= n \cdot (n-1) - (n-1) \cdot (n-2) + \bar{I}(n-1) + \bar{I}(n-1) \\ &= 2 \cdot (n-1) + 2 \cdot \bar{I}(n-1) \end{aligned}$$

Then we get:

$$n \cdot \bar{I}(n) - (n+1) \cdot \bar{I}(n-1) = 2 \cdot (n-1)$$

And if we divide both sides by $n \cdot (n+1)$, we get:

$$\frac{\bar{I}(n)}{n+1} - \frac{\bar{I}(n-1)}{n} = \frac{2 \cdot (n-1)}{n \cdot (n+1)}$$

Now let $g(n) = \frac{\bar{I}(n)}{n+1}$ and the equation becomes:

$$g(n) = g(n-1) + \frac{2 \cdot (n-1)}{n \cdot (n+1)} = \sum_{i=1}^{n} \left(\frac{2 \cdot (n-1)}{n \cdot (n+1)} \right)$$

Since we are interested in asymptotic behaviour, we can consider i, $i-1$ and $i+1$ as the same value when i is sufficiently high. Hence we have that:

$$g(n) \text{ is } \Theta \left(\sum_{i=1}^{n} \frac{1}{i} \right)$$

Figure 6 shows the sum that $g(n)$ is computing. It corresponds to an approximation of the integral of the function $\frac{1}{n}$ which is $\log(n)$. This sum is called a harmonic number and is denoted H_n. Hence we have:

$$g(n) \text{ is } \Theta \left(\log n \right)$$

Recalling that $g(n) = \frac{\bar{I}(n)}{n+1}$, we conclude that:

$$\bar{I}(n) \text{ is } \Theta \left(n \cdot \log n \right)$$

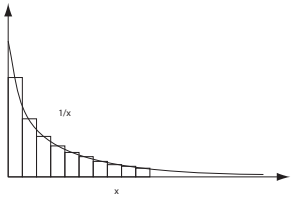

Figure 6: Approximation of the integral of $\frac{1}{x}$

We just derived the average complexity of the tree sort algorithm. Quicksort has the same average complexity. Here are a few remarks concerning average case analysis.

- The analysis is much more difficult than for the worst-case scenario.
- The result depends on the assumptions we made about the input, which might not be true in practice.

One solution to avoid the second problem is to use randomized algorithms. We will study randomized algorithms later in the semester, but a very simple approach would be to shuffle the list to be sorted before applying the sorting algorithm. Thus we would be given a truly uniform distribution of the input.

It is important to realize that the average running time of an algorithm does not tell everything we would like to know about the algorithm. Figures 7 and 8 show two different possible distributions of the running time which will both lead to an expected value of $n \cdot \log n$. However, in figure 7 there are a lot of cases in which the algorithm runs in $O\left(n^2\right)$. This is called a fat tail. On the other hand, in figure 8, only a few case run in $O\left(n^2\right)$. We call this a thin tail distribution. Obviously we would rather prefer having a thin tail than a fat tail.

2 Amortized Analysis

It is very common in algorithms that the data structure is initialized at the start of the algorithm, is accessed and modified during the life of the algorithm,

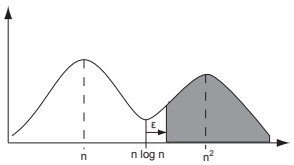

Figure 7: Fat tail distribution

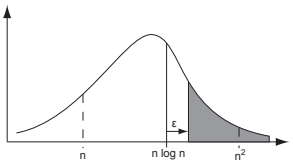

Figure 8: Thin tail distribution

Figure 9: Binary heap

and is cleaned up at the end. This gives the idea to do an amortized analysis of a data structure, that is to analyze the sum of the sequence of operations over the life of the structure rather than analyzing each operation on its own.

2.1 Stack

As a first, simple, example, let us consider a stack. A stack is a primitive data structure which has only two operations:

- Push
- Pop

Each of these operations can be easily implemented in $O(1)$ complexity. Now we would like to add a `Zero` operation, which simply resets the stack. If we implement it using `Push` and `Pop` operations (actually we only need `Pop` in this case), the resulting operation will run in $O(n)$ time.

However, if we consider the number of times `zero` is called over the lifetime of the structure with respect to the numbers of call to `Pop` and Push, we realize that `Push` need be called $O(n)$ times more. We say that `zero` is amortized over time to $O(1)$.

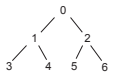

Figure 10: Binary heap tree

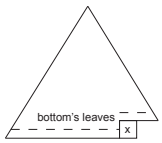

Figure 11: Insertion in a heap

2.2 Heap

As a more interesting example, we will now consider the *binary heap* data structure. A binary heap is a tree-based data structure, usually implemented using an array (Figure 9). The array at index 0 represents the root of the tree and then the next two cells are used to represent the left and right children of the root. And so on, each new cell is used to represent the first available children (Figure 10 shows which indices map to which nodes).

A heap must respect the *heap property*, which requires that the value of any node is smaller than the value of any children of that node. This ensures that any path from the root to a leaf goes through a sorted list of values.

A heap is useful to represent a priority queue, which supports two main operations:

- Insert
- DeleteMin

To do an Insert, we add the element at the last position in the array (which corresponds to the rightmost child, as shown in Figure 11). Then we need to check that the heap property is maintained. Only the new child could violate the property, so we test it against its parent. If it is smaller then we are done, if not we exchange the two and do the test again with the new parent. At best the element will stay at the end and it will take $O(1)$ time, at worst we will have to push the element all the way to the top, and this will take time $\Theta(\log n)$.

To do a DeleteMin, we need to remove the root—the first element of the array. Then, to avoid shifting the array, we take the last element of the array and put it at the beginning of the array. We then need to test the heap property and push the element down until the heap property is verified. Again this can take from $\Theta(1)$ to $\Theta(\log n)$.

With these implementations of Insert and DeleteMin, we have a data structure with a worst case complexity of $O(\log n)$.

We now consider an operation Meld that consists in merging two heaps. Using the array-based implementation, since we will need to go through the whole array, this operation will take at least $\Omega(n)$. However, we will see that it is possible to get $O(\log n)$ if we use a linked data structure.

In this new implementation, the heap will be a linked binary tree with the following properties:

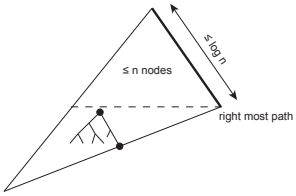

Figure 12: Leftist tree

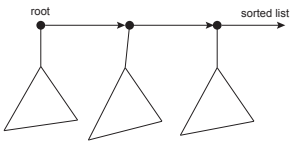

Figure 13: Sorted list

- It obeys the heap property.
- It obeys the leftist property.

A tree is leftist if, from any node, the shortest path to a leaf is the rightmost path (Figure 12).

Here is the main idea of the algorithm for the Meld. Because of the heap property, any path from the root to a leaf will be sorted. If we consider the rightmost path, we can view it as a sorted list, with each node having a pointer to its left subtree (Figure 13).

We can then merge two heaps by merging their rightmost paths. We remark that the rightmost path from the root cannot be longer than $\log n$. Since merging two sorted lists is linear in the size of the lists, the merge operation is $O(\log n)$. The resulting tree will still obey the heap property. However it may no longer be leftist: we need to do a bit more work to maintain the leftist property.