Chapter 4

Sorting

For the next series of lectures, we will focus on sorting. There a number of reasons for sorting. Here are a few important ones. Procedures for sorting are parts of many large software systems. Design of efficient sorting algorithms is necessary to achieve overall efficiency of these systems.

Sorting is well studied problem from the analysis point of view. Sorting is one of the few problems where provable lower bounds exist on how fast we can sort. In sorting, we are given an array $A[1..n]$ of n numbers We are to reorder these elements into increasing (or decreasing) order.

More generally, A is an array of objects and we sort them based on one of the attributes - the *key value*. The key value need not be a number. It can be any object from a *totally ordered domain*. Totally ordered domain means that for any two elements of the domain, x and y, either $x < y$, $x = y$ or $x > y$.

4.1 Slow Sorting Algorithms

There are a number of well-known slow $O(n^2)$ sorting algorithms. These include the following:

Bubble sort: Scan the array. Whenever two consecutive items are found that are out of order, swap them. Repeat until all consecutive items are in order.

Insertion sort: Assume that $A[1..i-1]$ have already been sorted. Insert $A[i]$ into its proper position in this sub array. Create this position by shifting all larger elements to the right.

Selection sort: Assume that $A[1..i-1]$ contain the $i-1$ smallest elements in sorted order. Find the smallest element in $A[i..n]$ Swap it with $A[i]$.

These algorithms are easy to implement. But they run in $\Theta(n^2)$ time in the worst case.

4.2 Sorting in $O(n \log n)$ time

We have already seen that Mergesort sorts an array of numbers in $\Theta(n \log n)$ time. We will study two others: *Heapsort* and *Quicksort*.

4.2.1 Heaps

A *heap* is a left-complete binary tree that conforms to the *heap order*. The heap order property: in a (min) heap, for every node X, the key in the parent is smaller than or equal to the key in X. In other words, the parent node has key smaller than or equal to both of its children nodes. Similarly, in a max heap, the parent has a key larger than or equal both of its children Thus the smallest key is in the root in a min heap; in the max heap, the largest is in the root.

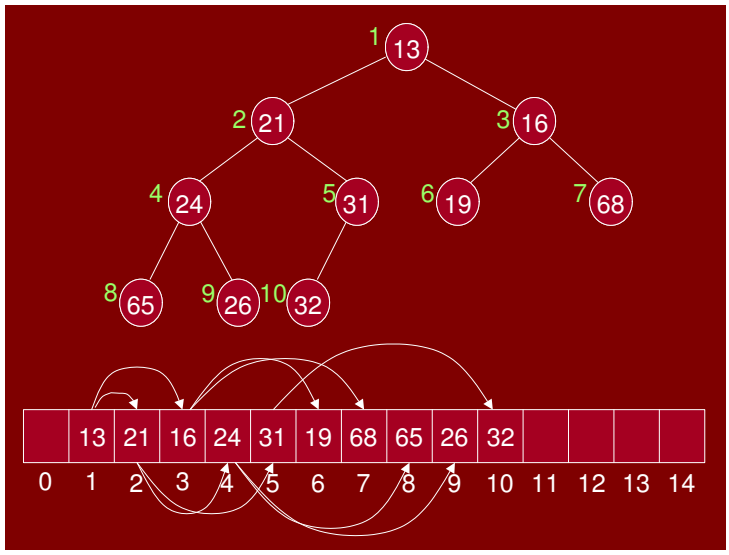

Figure 4.1: A min-heap

The number of nodes in a complete binary tree of height h is

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^h = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

h in terms of n is

$$h = (\log(n+1)) - 1 \approx \log n \in \Theta(\log n)$$

One of the clever aspects of heaps is that they can be stored in arrays without using any pointers. This is due to the left-complete nature of the binary tree. We store the tree nodes in level-order traversal. Access

to nodes involves simple arithmetic operations:

$$\begin{aligned} \texttt{left}(i) &: \text{returns } 2i, \text{ index of left child of node } i. \\ \texttt{right}(i) &: \text{returns } 2i+1, \text{ the right child.} \\ \texttt{parent}(i) &: \text{returns } \lfloor i/2 \rfloor, \text{ the parent of } i. \end{aligned}$$

The root is at position 1 of the array.

4.2.2 Heapsort Algorithm

We build a max heap out of the given array of numbers $A[1..n]$. We repeatedly extract the the maximum item from the heap. Once the max item is removed, we are left with a hole at the root. To fix this, we will replace it with the last leaf in tree. But now the heap order will very likely be destroyed. We will apply a heapify procedure to the root to restore the heap. Figure 4.2 shows an array being sorted.

```
HEAPSORT( array A, int n)
1  BUILD-HEAP(A, n)
2  m ← n
3  while (m ≥ 2)
4  do SWAP(A[1], A[m])
5     m ← m − 1
6     HEAPIFY(A, 1, m)
```

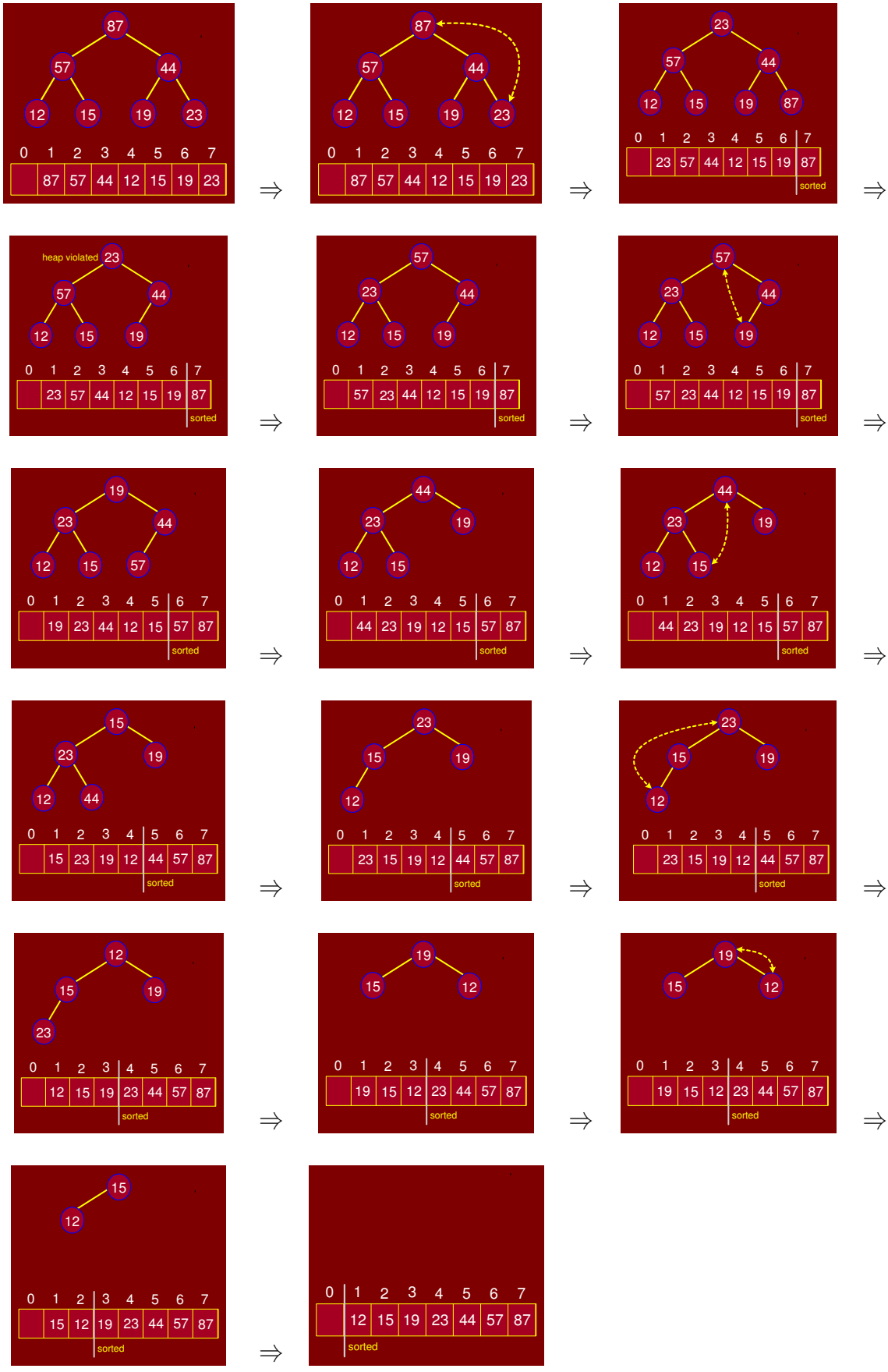

Figure 4.2: Example of heap sort

4.2.3 Heapify Procedure

There is one principal operation for maintaining the heap property. It is called Heapify. (In other books it is sometimes called sifting down.) The idea is that we are given an element of the heap which we suspect may not be in valid heap order, but we assume that all of other the elements in the subtree rooted at this element are in heap order. In particular this root element may be too small. To fix this we "sift" it down the tree by swapping it with one of its children. Which child? We should take the larger of the two children to satisfy the heap ordering property. This continues recursively until the element is either larger than both its children or until its falls all the way to the leaf level. Here is the algorithm. It is given the heap in the array A, and the index i of the suspected element, and m the current active size of the heap. The element $A[max]$ is set to the maximum of $A[i]$ and it two children. If $max \neq i$ then we swap $A[i]$ and $A[max]$ and then recurse on $A[max]$.

```
HEAPIFY( array A, int i, int m)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  max ← i
 4  if (l ≤ m)and(A[l] > A[max])
 5     then max ← l
 6  if (r ≤ m)and(A[r] > A[max])
 7     then max ← r
 8  if (max ≠ i)
 9     then SWAP(A[i], A[max])
10          HEAPIFY(A, max, m)
```

4.2.4 Analysis of Heapify

We call heapify on the root of the tree. The maximum levels an element could move up is $\Theta(\log n)$ levels. At each level, we do simple comparison which $O(1)$. The total time for heapify is thus $O(\log n)$. Notice that it is not $\Theta(\log n)$ since, for example, if we call heapify on a leaf, it will terminate in $\Theta(1)$ time.

4.2.5 BuildHeap

We can use Heapify to build a heap as follows. First we start with a heap in which the elements are not in heap order. They are just in the same order that they were given to us in the array A. We build the heap by starting at the leaf level and then invoke Heapify on each node. (Note: We cannot start at the top of the tree. Why not? Because the precondition which Heapify assumes is that the entire tree rooted at node i is already in heap order, except for i.) Actually, we can be a bit more efficient. Since we know that each leaf is already in heap order, we may as well skip the leaves and start with the first non-leaf node. This will be in position $n/2$. (Can you see why?)

Here is the code. Since we will work with the entire array, the parameter m for Heapify, which indicates the current heap size will be equal to n, the size of array A, in all the calls.

```
BUILDHEAP( array A, int n)
1  for i ← n/2 downto 1
2  do
3      HEAPIFY(A, i, n)
```

4.2.6 Analysis of BuildHeap

For convenience, we will assume $n = 2^{h+1} - 1$ where h is the height of tree. The heap is a left-complete binary tree. Thus at each level j, $j < h$, there are 2^j nodes in the tree. At level h, there will be 2^h or less nodes. How much work does buildHeap carry out? Consider the heap in Figure 4.3:

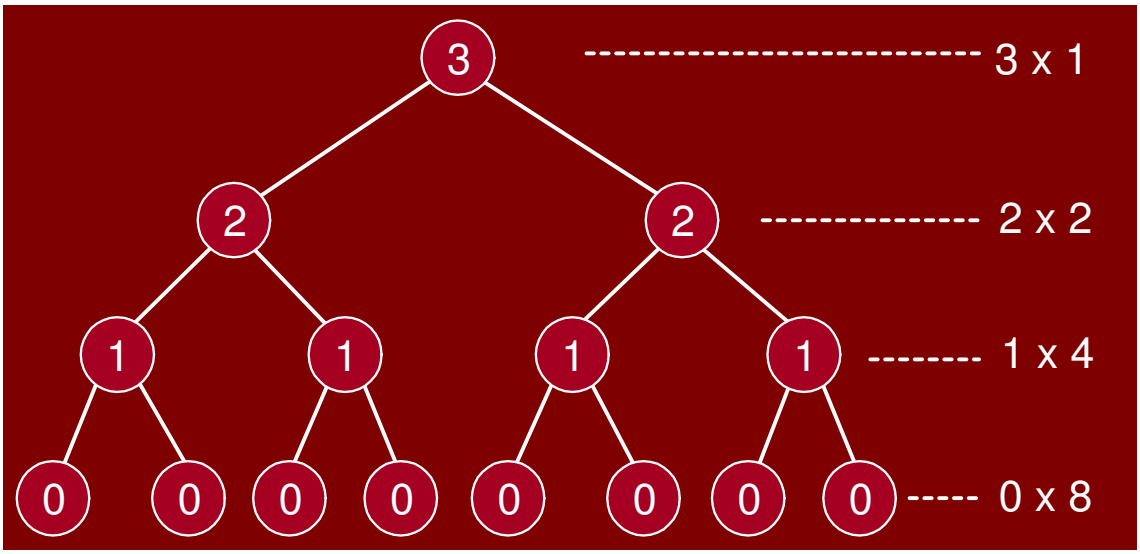

Figure 4.3: Total work performed in buildheap

At the bottom most level, there are 2^h nodes but we do not heapify these. At the next level up, there are 2^{h-1} nodes and each might shift down 1. In general, at level j, there are 2^{h-j} nodes and each may shift down j levels.

So, if count from bottom to top, level-by-level, the total time is

$$T(n) = \sum_{j=0}^{h} j2^{h-j} = \sum_{j=0}^{h} j\frac{2^h}{2^j}$$

We can factor out the 2^h term:

$$T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j}$$

How do we solve this sum? Recall the geometric series, for any constant $x < 1$

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}$$

Take the derivative with respect to x and multiply by x

$$\sum_{j=0}^{\infty} jx^{j-1} = \frac{1}{(1-x)^2} \qquad \sum_{j=0}^{\infty} jx^j = \frac{x}{(1-x)^2}$$

We plug $x = 1/2$ and we have the desired formula:

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{1/2}{(1-(1/2))^2} = \frac{1/2}{1/4} = 2$$

In our case, we have a bounded sum, but since we the infinite series is bounded, we can use it as an easy approximation:

$$\begin{aligned} T(n) &= 2^h \sum_{j=0}^{h} \frac{j}{2^j} \\ &\leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \\ &\leq 2^h \cdot 2 = 2^{h+1} \end{aligned}$$

Recall that $n = 2^{h+1} - 1$. Therefore

$$T(n) \leq n + 1 \in O(n)$$

The algorithm takes at least $\Omega(n)$ time since it must access every element at once. So the total time for BuildHeap is $\Theta(n)$.

BuildHeap is a relatively complex algorithm. Yet, the analysis yield that it takes $\Theta(n)$ time. An intuitive way to describe why it is so is to observe an important fact about binary trees The fact is that the vast majority of the nodes are at the lowest level of the tree. For example, in a complete binary tree of height h, there is a total of $n \approx 2^{h+1}$ nodes.

The number of nodes at the bottom three levels alone is

$$2^h + 2^{h-1} + 2^{h-2} = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} = \frac{7n}{8} = 0.875n$$

Almost 90% of the nodes of a complete binary tree reside in the 3 lowest levels. Thus, algorithms that operate on trees should be efficient (as BuildHeap is) on the bottom-most levels since that is where most of the weight of the tree resides.

4.2.7 Analysis of Heapsort

Heapsort calls BuildHeap once. This takes $\Theta(n)$. Heapsort then extracts roughly n maximum elements from the heap. Each extract requires a constant amount of work (swap) and $O(\log n)$ heapify. Heapsort is thus $O(n \log n)$.

Is HeapSort $\Theta(n \log n)$? The answer is yes. In fact, later we will show that comparison based sorting algorithms can not run faster than $\Omega(n \log n)$. Heapsort is such an algorithm and so is Mergesort that we saw ealier.

4.3 Quicksort

Our next sorting algorithm is Quicksort. It is one of the fastest sorting algorithms known and is the method of choice in most sorting libraries. Quicksort is based on the divide and conquer strategy. Here is the algorithm:

```
QUICKSORT( array A, int p, int r)
1  if (r > p)
2    then
3         i ← a random index from [p..r]
4          swap A[i] with A[p]
5         q ← PARTITION(A, p, r)
6         QUICKSORT(A, p, q − 1)
7         QUICKSORT(A, q + 1, r)
```

4.3.1 Partition Algorithm

Recall that the partition algorithm partitions the array $A[p..r]$ into three sub arrays about a pivot element x. $A[p..q-1]$ whose elements are less than or equal to x, $A[q] = x$, $A[q+1..r]$ whose elements are greater than x

We will choose the first element of the array as the pivot, i.e. $x = A[p]$. If a different rule is used for selecting the pivot, we can swap the chosen element with the first element. We will choose the pivot randomly.

The algorithm works by maintaining the following *invariant condition*. $A[p] = x$ is the pivot value. $A[p..q-1]$ contains elements that are less than x, $A[q+1..s-1]$ contains elements that are greater than

or equal to x $A[s..r]$ contains elements whose values are currently unknown.

```
PARTITION( array A, int p, int r)
1  x ← A[p]
2  q ← p
3  for s ← p + 1 to r
4  do if (A[s] < x)
5        then q ← q + 1
6             swap A[q] with A[s]
7   swap A[p] with A[q]
8  return q
```

Figure 4.4 shows the execution trace partition algorithm.

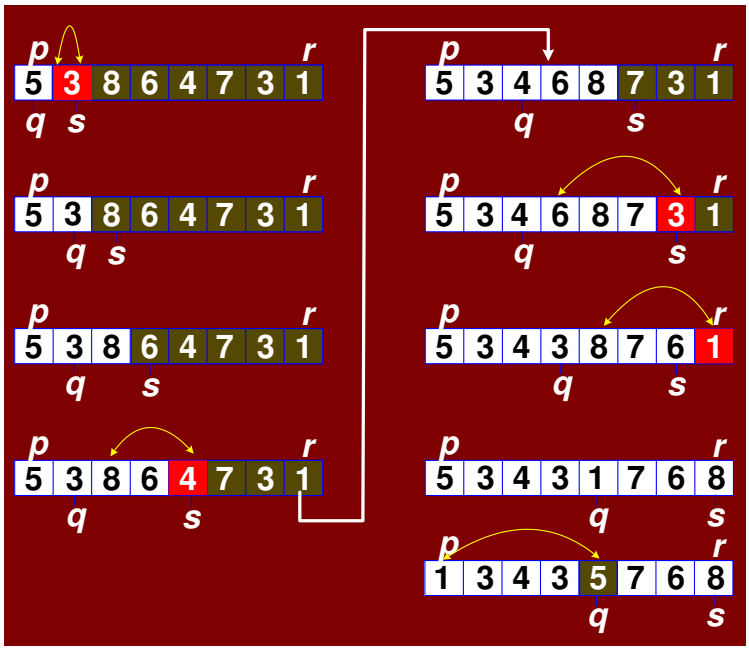

Figure 4.4: Trace of partitioning algorithm

4.3.2 Quick Sort Example

The Figure 4.5 trace out the quick sort algorithm. The first partition is done using the last element, 10, of the array. The left portion are then partitioned about 5 while the right portion is partitioned about 13. Notice that 10 is now at its final position in the eventual sorted order. The process repeats as the algorithm recursively partitions the array eventually sorting it.

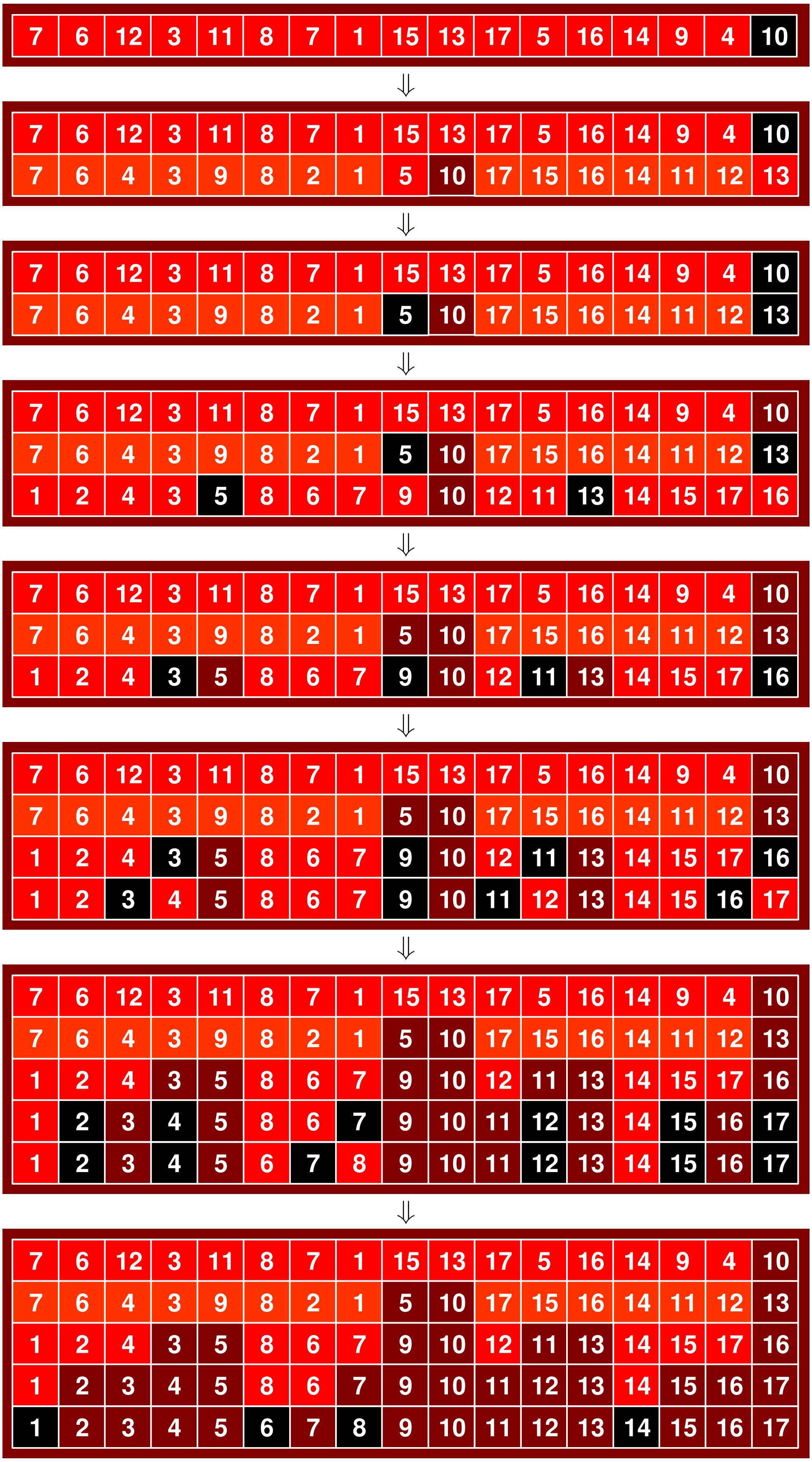

Figure 4.5: Example of quick sort

It is interesting to note (but not surprising) that the pivots form a binary search tree. This is illustrated in Figure 4.6.

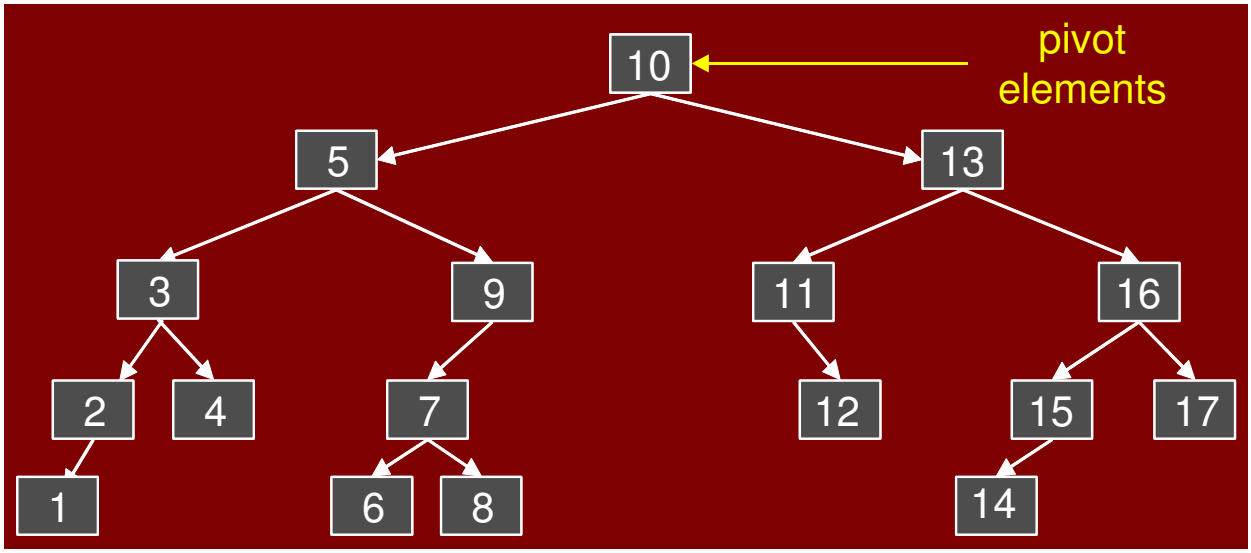

Figure 4.6: Quicksort BST

4.3.3 Analysis of Quicksort

The running time of quicksort depends heavily on the selection of the pivot. If the rank of the pivot is very large or very small then the partition (BST) will be unbalanced. Since the pivot is chosen randomly in our algorithm, the expected running time is $O(n \log n)$. The worst case time, however, is $O(n^2)$. Luckily, this happens rarely.

4.3.4 Worst Case Analysis of Quick Sort

Let's begin by considering the worst-case performance, because it is easier than the average case. Since this is a recursive program, it is natural to use a recurrence to describe its running time. But unlike MergeSort, where we had control over the sizes of the recursive calls, here we do not. It depends on how the pivot is chosen. Suppose that we are sorting an array of size n, $A[1 : n]$, and further suppose that the pivot that we select is of rank q, for some q in the range 1 to n. It takes $\Theta(n)$ time to do the partitioning and other overhead, and we make two recursive calls. The first is to the subarray $A[1 : q - 1]$ which has $q - 1$ elements, and the other is to the subarray $A[q + 1 : n]$ which has $n - q$ elements. So if we ignore the $\Theta(n)$ (as usual) we get the recurrence:

$$T(n) = T(q - 1) + T(n - q) + n$$

This depends on the value of q. To get the worst case, we maximize over all possible values of q. Putting is together, we get the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \max\limits_{1 \leq q \leq n} (T(q-1) + T(n-q) + n) & \text{otherwise} \end{cases}$$

Recurrences that have max's and min's embedded in them are very messy to solve. The key is determining which value of q gives the maximum. (A rule of thumb of algorithm analysis is that the worst cases tends to happen either at the extremes or in the middle. So I would plug in the value $q = 1$, $q = n$, and $q = n/2$ and work each out.) In this case, the worst case happens at either of the extremes. If we expand the recurrence for $q = 1$, we get:

$$\begin{aligned} T(n) &\leq T(0) + T(n-1) + n \\ &= 1 + T(n-1) + n \\ &= T(n-1) + (n+1) \\ &= T(n-2) + n + (n+1) \\ &= T(n-3) + (n-1) + n + (n+1) \\ &= T(n-4) + (n-2) + (n-1) + n + (n+1) \\ &= T(n-k) + \sum_{i=-1}^{k-2} (n-i) \end{aligned}$$

For the basis $T(1) = 1$ we set $k = n - 1$ and get

$$\begin{aligned} T(n) &\leq T(1) + \sum_{i=-1}^{n-3} (n-i) \\ &= 1 + (3 + 4 + 5 + \cdots + (n-1) + n + (n+1)) \\ &\leq \sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2} \in \Theta(n^2) \end{aligned}$$

4.3.5 Average-case Analysis of Quicksort

We will now show that in the average case, quicksort runs in $\Theta(n \log n)$ time. Recall that when we talked about average case at the beginning of the semester, we said that it depends on some assumption about the distribution of inputs. However, in the case of quicksort, the analysis does not depend on the distribution of input at all. It only depends upon the random choices of pivots that the algorithm makes. This is good, because it means that the analysis of the algorithm's performance is the same for all inputs. In this case the average is computed over all possible random choices that the algorithm might make for the choice of the pivot index in the second step of the QuickSort procedure above.

To analyze the average running time, we let $T(n)$ denote the average running time of QuickSort on a list of size n. It will simplify the analysis to assume that all of the elements are distinct. The algorithm has n

random choices for the pivot element, and each choice has an equal probability of $1/n$ of occuring. So we can modify the above recurrence to compute an average rather than a max, giving:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \frac{1}{n}\sum_{q=1}^{n}(T(q-1)+T(n-q)+n) & \text{otherwise} \end{cases}$$

The time $T(n)$ is the weighted sum of the times taken for various choices of q. I.e.,

$$\begin{aligned} T(n) = \Big[&\frac{1}{n}(T(0)+T(n-1)+n) \\ &+\frac{1}{n}(T(1)+T(n-2)+n) \\ &+\frac{1}{n}(T(2)+T(n-3)+n) \\ &+\cdots+\frac{1}{n}(T(n-1)+T(0)+n)\Big] \end{aligned}$$

We have not seen such a recurrence before. To solve it, expansion is possible but it is rather tricky. We will attempt a constructive induction to solve it. We know that we want a $\Theta(n \log n)$. Let us assume that $T(n) \leq cn \log n$ for $n \geq 2$ where c is a constant.

For the base case $n = 2$ we have

$$\begin{aligned} T(n) &= \frac{1}{2}\sum_{q=1}^{2}(T(q-1)+T(2-q)+2) \\ &= \frac{1}{2}\Big[(T(0)+T(1)+2)+(T(1)+T(0)+2)\Big] \\ &= \frac{8}{2} = 4. \end{aligned}$$

We want this to be at most $c2\log 2$, i.e.,

$$T(2) \leq c2\log 2$$

or

$$4 \leq c2\log 2$$

therefore

$$c \geq 4/(2\log 2) \approx 2.88.$$

For the induction step, we assume that $n \geq 3$ and The induction hypothesis is that for any $n' < n$, we have $T(n') \leq cn' \log n'$. We want to prove that it is true for $T(n)$. By expanding $T(n)$ and moving the

factor of n outside the sum we have

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{q=1}^{n}(T(q-1)+T(n-q)+n) \\ &= \frac{1}{n}\sum_{q=1}^{n}(T(q-1)+T(n-q))+n \\ &= \frac{1}{n}\sum_{q=1}^{n}T(q-1)+\frac{1}{n}\sum_{q=1}^{n}T(n-q)+n \end{aligned}$$

$$T(n) = \frac{1}{n}\sum_{q=1}^{n}T(q-1)+\frac{1}{n}\sum_{q=1}^{n}T(n-q)+n$$

Observe that the two sums add up the same values $T(0)+T(1)+\cdots+T(n-1)$. One counts up and other counts down. Thus we can replace them with $2\sum_{q=0}^{n-1}T(q)$. We will extract $T(0)$ and $T(1)$ and treat them specially. These two do not follow the formula.

$$\begin{aligned} T(n) &= \frac{2}{n}\Big(\sum_{q=0}^{n-1}T(q)\Big)+n \\ &= \frac{2}{n}\Big(T(0)+T(1)+\sum_{q=2}^{n-1}T(q)\Big)+n \end{aligned}$$

We will apply the induction hypothesis for $q < n$ we have

$$\begin{aligned} T(n) &= \frac{2}{n}\Big(T(0)+T(1)+\sum_{q=2}^{n-1}T(q)\Big)+n \\ &\leq \frac{2}{n}\Big(1+1+\sum_{q=2}^{n-1}(cq\log q)\Big)+n \\ &= \frac{2c}{n}\Big(\sum_{q=2}^{n-1}(cq\ln q)\Big)+n+\frac{4}{n} \end{aligned}$$

We have never seen this sum before:

$$S(n) = \sum_{q=2}^{n-1}(cq\ln q)$$

Recall from calculus that for any monotonically increasing function $f(x)$

$$\sum_{i=a}^{b-1}f(i) \leq \int_{a}^{b}f(x)dx$$

The function $f(x) = x\ln x$ is monotonically increasing, and so

$$S(n) = \sum_{q=2}^{n-1} (cq\ln q) \leq \int_2^n x\ln x\, dx \tag{4.1}$$

We can integrate this by parts:

$$\int_2^n x\ln x\, dx = \frac{x^2}{2}\ln x - \frac{x^2}{4}\Big|_{x=2}^{n}$$

$$\begin{aligned}\int_2^n x\ln x\, dx &= \frac{x^2}{2}\ln x - \frac{x^2}{4}\Big|_{x=2}^{n} \\ &= \Big(\frac{n^2}{2}\ln n - \frac{n^2}{4}\Big) - (2\ln 2 - 1) \\ &\leq \frac{n^2}{2}\ln n - \frac{n^2}{4}\end{aligned}$$

We thus have

$$S(n) = \sum_{q=2}^{n-1} (cq\ln q) \leq \frac{n^2}{2}\ln n - \frac{n^2}{4}$$

Plug this back into the expression for $T(n)$ to get

$$T(n) = \frac{2c}{n}\Big(\frac{n^2}{2}\ln n - \frac{n^2}{4}\Big) + n + \frac{4}{n}$$

$$\begin{aligned}T(n) &= \frac{2c}{n}\Big(\frac{n^2}{2}\ln n - \frac{n^2}{4}\Big) + n + \frac{4}{n} \\ &= cn\ln n - \frac{cn}{2} + n + \frac{4}{n} \\ &= cn\ln n + n\big(1 - \frac{c}{2}\big) + \frac{4}{n}\end{aligned}$$

$$T(n) = cn\ln n + n\big(1 - \frac{c}{2}\big) + \frac{4}{n}$$

To finish the proof,we want all of this to be at most $cn\ln n$. For this to happen, we will need to select c such that

$$n\big(1 - \frac{c}{2}\big) + \frac{4}{n} \leq 0$$

If we select $c = 3$, and use the fact that $n \geq 3$ we get

$$\begin{aligned}n\big(1 - \frac{c}{2}\big) + \frac{4}{n} &= \frac{3}{n} - \frac{n}{2} \\ &\leq 1 - \frac{3}{2} = -\frac{1}{2} \leq 0.\end{aligned}$$

From the basis case we had $c \geq 2.88$. Choosing $c = 3$ satisfies all the constraints. Thus $T(n) = 3n\ln n \in \Theta(n\log n)$.

4.4 In-place, Stable Sorting

An *in-place* sorting algorithm is one that uses no additional array for storage. A sorting algorithm is *stable* if duplicate elements remain in the same relative position after sorting.

$9 \mid 3 \mid 3' \mid 5 \mid 6 \mid 5' \mid 2 \mid 1 \mid 3''$	unsorted
$1 \mid 2 \mid 3 \mid 3' \mid 3'' \mid 5 \mid 5' \mid 6 \mid 9$	stable sort
$1 \mid 2 \mid 3' \mid 3 \mid 3'' \mid 5' \mid 5 \mid 6 \mid 9$	unstable

Bubble sort, insertion sort and selection sort are in-place sorting algorithms. Bubble sort and insertion sort can be implemented as stable algorithms but selection sort cannot (without significant modifications). Mergesort is a stable algorithm but not an in-place algorithm. It requires extra array storage. Quicksort is not stable but is an in-place algorithm. Heapsort is an in-place algorithm but is not stable.

4.5 Lower Bounds for Sorting

The best we have seen so far is $O(n \log n)$ algorithms for sorting. Is it possible to do better than $O(n \log n)$? If a sorting algorithm is solely based on comparison of keys in the array then it is *impossible* to sort more efficiently than $\Omega(n \log n)$ time. All algorithms we have seen so far are comparison-based sorting algorithms.

Consider sorting three numbers a_1, a_2, a_3. There are $3! = 6$ possible combinations:

$$(a_1, a_2, a_3), (a_1, a_3, a_2), (a_3, a_2, a_1)$$
$$(a_3, a_1, a_2), (a_2, a_1, a_3), (a_2, a_3, a_1)$$

One of these permutations leads to the numbers in sorted order.

The comparison based algorithm defines a *decision tree*. Here is the tree for the three numbers.

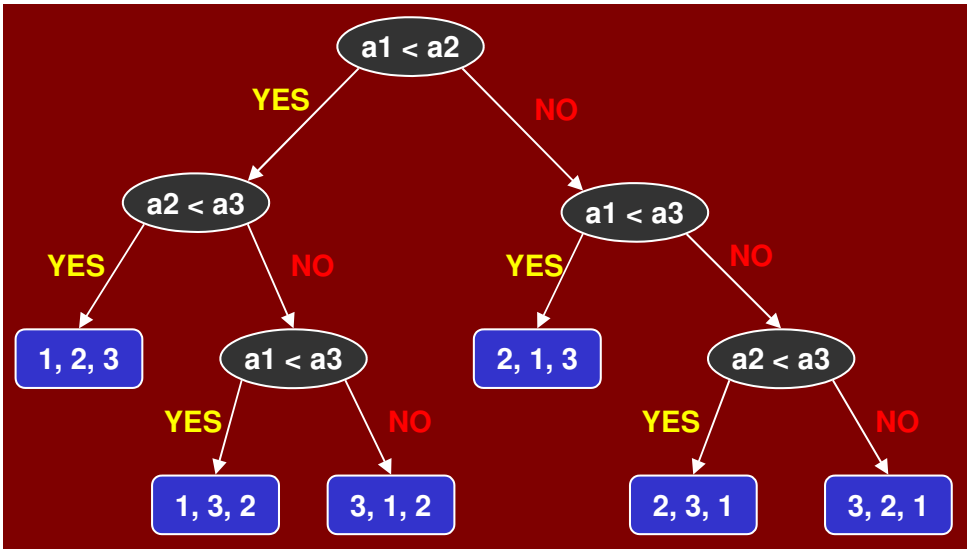

Figure 4.7: Decision Tree

For n elements, there will be $n!$ possible permutations. The height of the tree is exactly equal to $T(n)$, the running time of the algorithm. The height is $T(n)$ because any path from the root to a leaf corresponds to a sequence of comparisons made by the algorithm.

Any binary tree of height $T(n)$ has at most $2^{T(n)}$ leaves. Thus a comparison based sorting algorithm can distinguish between at most $2^{T(n)}$ different final outcomes. So we have

$$\begin{aligned} 2^{T(n)} &\geq n! \qquad \text{and therefore} \\ T(n) &\geq \log(n!) \end{aligned}$$

We can use *Stirling's approximation* for $n!$:

$$n! \geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Thereofore

$$\begin{aligned} T(n) &\geq \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) \\ &= \log(\sqrt{2\pi n}) + n\log n - n\log e \\ &\in \Omega(n\log n) \end{aligned}$$

We thus have the following theorem.

Theorem 1
Any comparison-based sorting algorithm has worst-case running time $\Omega(n\log n)$.