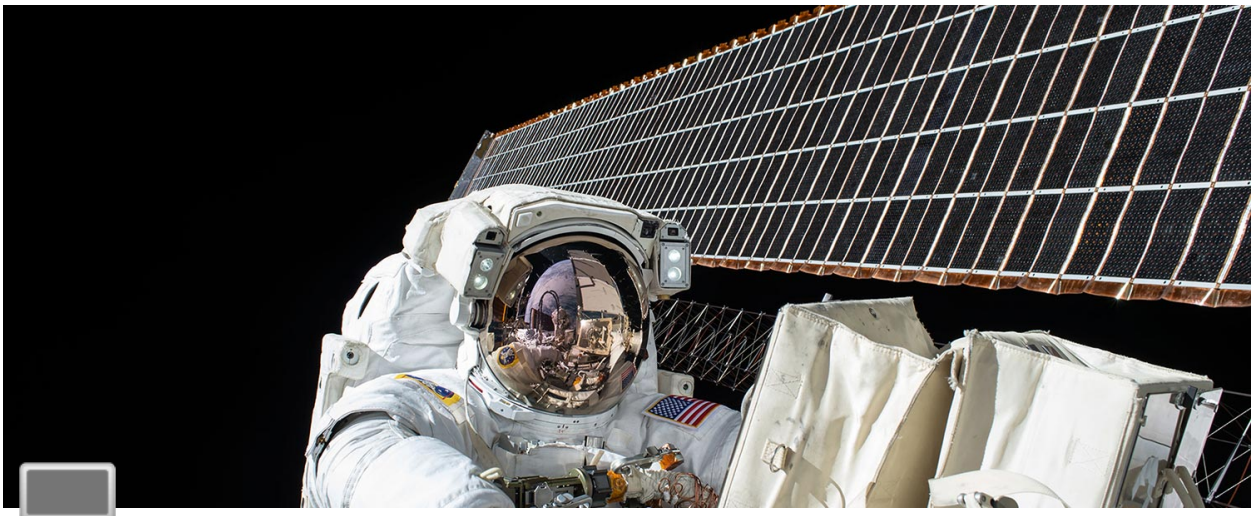
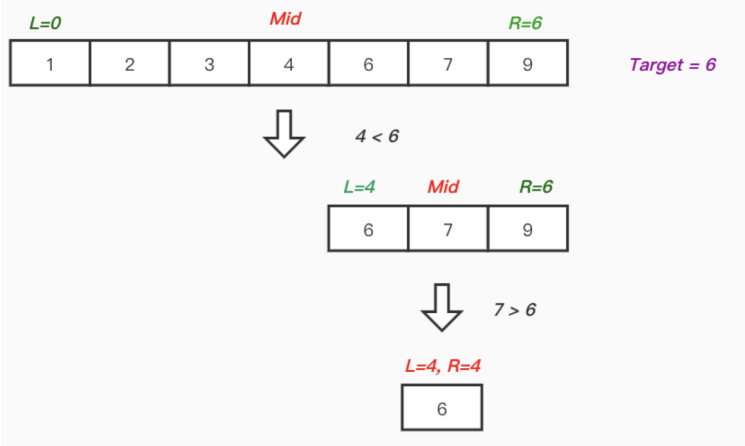


00_CS350 Algorithms Exam 1

Ryan Filgas - studying under professor David Ely

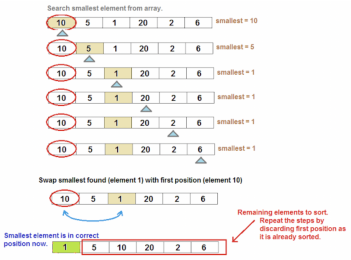
Linear Search:	Check each index. Return matching index.
Space Efficiency	O(1) - In place algorithm.
Time Efficiency (Big O) - “At worst”	O(n) - 1 compare at each index
Big Omega Ω - “At Best” - Lower Bound	O(1)
Big Theta Θ - Tight upper and lower bound.	—

Binary Search:	Given a sorted list, search it by successive guesses.
Space Efficiency	O(1) - In place algorithm
Time Efficiency (Big O) - Worst Case	log(n) - The search is unsuccessful.
Big Omega Ω - “At best”	O(1)
Big Theta Θ - Tight upper and lower bound.	—




```
// The binary search algorithm works as follows. Given an array of 'n' sorted integers,
// guess thelocation of a number at the index n/2. If the number is bigger, set the
// bottom of the curent list at ('bottom' + 'top')/2. If the number is larger, set the
// top to bottom + top / 2. Guess a new number at bottom + top / 2. Rinse and repeat
// until 'bottom' and 'top' are the same.
```

Selection Sort:	Swap smaller items to the front until completion.
Space Efficiency	O(1) - In place algorithm
Time Efficiency (Big O) - Worst Case	O(n^2) - Sorted largest to smallest. [n(n+1)]/2 compares and swaps.
Big Omega Ω - “At best”	Ω(n^2) - [n(n+1)]/2 compares and 0 swaps.
Big Theta Θ - Tight upper and lower bound	—




```
// The selection sort algorithm proceeds as follows: start base pointer at index 0. Start
// the offset pointer at index 1. If its value is smaller than the offset, swap base value
// with offset value. Continue doing this until the end of the array. Advance the base
// value and repeat. Continue until the end of the array has been reached.
 15 void sort(int array[] int size){
 16     //lf = left, rf = right
 17     for(lf = 0; lf < n - 1; lf++){
 18         for(rf = lf + 1; rf < n; rf++){
 19             if(x[rf] < x[lf]){
 20                 temp = x[lf];
 21                 x[lf] = x[rf];
 22                 x[rf] = temp;}}}} //ugly I know. Saving space.
```

Merge Sort:	Have children merge recursively, sort in base case.
Space Efficiency	O(n) - At most 2n memory will be in use.
Time Efficiency (Big O) - Worst Case	O(n*log(n)) - constant time to sort n items * log n calls to children.
Big Omega Ω - “At best - ”O(n*log(n))	O(n*log(n)) - worst case = best case.
Big Theta Θ - Tight upper and lower bound	O(n*log(n)) - if proven.

```
// The merge sort algorithm works as follows. Split the array and call your merge
// function recursively. Call it once on the left half, and once on the right half.
// When the base case is hit, check if the two values should be swapped, the smaller
// should go first. Return. After the recursive call, merge the two halves. With two
// pointers, p1 and p2, insert the greater of each into a new array and advance the
// pointer that was copied. Once the end is reached on one, copy the stragglers over
// from the other. Copy the sorted array back to the original. The mergesort is complete.

 71 void mergeSort(int array[], int left, int right){
 72     if(right == left)
 73         return;
 74     int middle = (left + right) / 2;
 76     //recursive left sort, inclusive of middle
 77     mergeSort(array, left, middle);
 78     //recursive right sort exclusive of middle
 79     mergeSort(array, middle + 1, right);
 80     //merge
 81     merge(array, left, middle, right);
 82     return;
 83 }
 87 void merge(int array[], int left, int middle, int right){
 88     //if we get down to two elements, swap as necessary.
 89     if(right  == left)
 90         return;
 91     int size = right - left + 1;
 92     int temparray[size];
 93     int current = 0;
 95     int l = left;
 96     int r = middle + 1;
 98     //take care of normal case. Everything finishes at the same time, so copy the smaller
 99     //values in incrementally until the sort is finished on at least one side.
100     while(l <= middle && r <= right){
101         if(array[l] <= array[r]){
102             temparray[current] = array[l];
103             ++l;
104         }
105         else{
106             temparray[current] = array[r];
107             ++r;
108         }
```

```
109         ++current;
110     }
112     //when one side finishes the merge before the other, copy the rest into the temp array.
113     while(l <= middle && current < size){
114         temparray[current] = array[l];
115         ++current;
116         ++l;
117     }
118     while(r <= right && current < size){
119         temparray[current] = array[r];
120         ++current;
121         ++r;
122     }
125     //Copy the result back to the main array.
126     for(int i = 0, j = left; i < size && j <= right; ++i, ++j){
127         array[j] = temparray[i];
128     }
129     return;
130 }
```

Heap Sort:	Build a max heap, steal from the top, heap again.
Space Efficiency	O(1) - In place algorithm
Time Efficiency (Big O) - Worst Case	O(n*log(n)) - log n layers with n compares/swaps
Big Omega Ω - “At best - ”O(n*log(n))	O(n*log(n)) - worst case = best case
Big Theta Θ - Tight upper and lower bound	O(n*log(n)) - if proven.
Efficiency of heap operation?	n for sift down, n log n for sift up

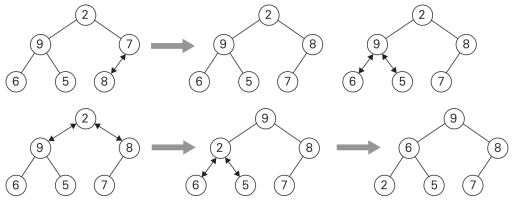


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

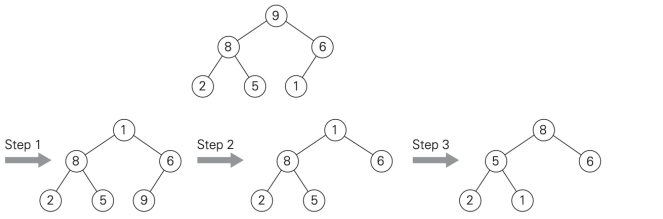


FIGURE 6.13 Deleting the root’s key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is “heapified” by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

> How efficient is this algorithm in the worst case? Assume, for simplicity, that n = 2k − 1 so that a heap’s tree is full, i.e., the largest possible number of

> nodes occurs on each level. Let h be the height of the tree. According to the first

> property of heaps in the list at the beginning of the section, h = ⌊log2 n⌋ or just
> ⌈log2 (n + 1)⌉ − 1 = k − 1 for the specific values of n we are considering. Each
> key on level i of the tree will travel to the leaf level h in the worst case of the heap construction algorithm. Since moving to the next level down requires two
> comparisons—one to find the larger child and the other to determine whether the exchange is required—the total number of key comparisons involving a key
> on level i will be 2(h − i). Therefore, the total number of key comparisons in the
> worst case will be

> h−1 h−1
> $Cworst(n) = \square\square 2(h-i) = \square 2(h-i)2i = 2(n-log2(n+1))$

> i=0 level i keys i=0

> where the validity of t□he last equality can be proved either by using the closed-form
> formula for the sum hi=1 i2i (see Appendix A) or by mathematical induction on
> h. Thus, with this bottom-up algorithm, a heap of size n can be constructed with
> fewer than 2n comparisons.

```
// The heapsort algorithm for a max heap works in one of two ways. The heap is
// represented by an array, but treated like a tree.
// Heap method 1 Sift up: When building the heap, each piece of data is "loaded" at the end
// of the line in the array like a queue. If it's larger than its parent, it will
// swap with them, and cause a chain reaction until the largest number is at the
// top of the heap.
```

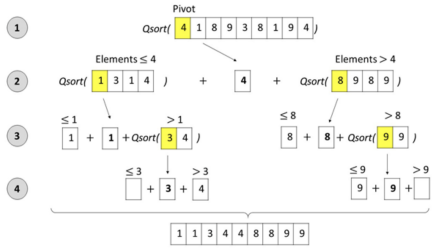
```
// Heap method 2 Sift down: Instead start adding at the parent of the last node, and if needed
// swap down. To do this recursively, call the heapify method on both children, then check
// their values and swap with the largest child

 63 // 1. Check each index, and swap with parents if they are smaller until root.
 64 void buildHeap(int arr[], int size){
 65     int top = 0;
 66     int current = 0;
 67     int parent = 0;
 68     do{
 69         //count points to the last "node". We need to check its parent for a swap.
 70         current = top; //set count as the current "node".
 71         ++top; //increment count for the next run
 72         parent = getParent(current); //get the index of the parent
 73         int exit = 0;
 74         while(!exit && (arr[current] > arr[parent])){
 75             if(parent == 0)
 76                 exit = 1; //This is the LAST swap. Time to leave the loop.
 77             swap(arr, current, parent);
 78             current = parent;
 79             parent = getParent(current);
 80         }
 81     }while(top < size);
 82 }

 86 // 1. Swap first with last.
 87 // 2. Decrement the end. We put the largest where it goes.
 88 // 3. Swap the smaller value with the larger of the left and right children down the chain.
 89 // 4. Rinse and repeat.
 90 // 5. Deal with base case at the end. Indexes 0 and 1 shouldn't assume a swap.
 91 void sortHeap(int arr[], int size){
 92     int end = size - 1;
 93     int start = 0;
 94     int parent = 0;
 95     int child = 0;//get the index of the greater of two children.
 96     int temp = 0;
 97     do{
 98         swap(arr, start, end);
 99         parent = 0; // Parent should always be the root starting the swapping process.
100         --end; //Decrement end. This works towards our base case.
101         child = greater(arr, getRightChild(parent), getLeftChild(parent), size);
102         while(child <= end && (arr[parent] < arr[child])){
103             swap(arr, child, parent);
104             temp = child;
105             child = greater(arr, getRightChild(child), getLeftChild(child), size);
106             parent = temp;
107         }
108     }while(end > 1); // Reached base case, there are two nodes left to compare.
110     //Fringe/Base case.
111     if(arr[0] > arr[1]){
112         swap(arr, 0, 1);
113     }
114     return;
115 }
119 int getParent(int child){
```

```
120     return ((child - 1)/2);
121 }
125 int getRightChild(int parent){
126     return((2 * parent) + 2);
127 }
130 int getLeftChild(int parent){
131     return((2 * parent) + 1);
132 }
```

Quick Sort:	
Space Efficiency	O(1) - In place algorithm
Time Efficiency (Big O) - Worst Case	O(n^2) - but O(nlogn) average
Big Omega Ω - “At best - ”O(n*log(n))	O(n*log(n))
Big Theta Θ - Tight upper and lower bound	—



Thus, the question about the utility of quicksort comes down to its average-case behavior. Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n. A partition can happen in any position s ($0 \le s \le n-1$) after $n+1$ comparisons are made to achieve the partition. After the partition, the left and right subarrays will have s and $n-1-s$ elements, respectively. Assuming that the partition split can happen in each position s with the same probability $1/n$, we get the following recurrence relation:

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$
$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

Its solution, which is much trickier than the worst- and best-case analyses, turns out to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$

In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. This unfortunate situation will happen, in particular, for increasing arrays, i.e., for inputs for which the problem is already solved! Indeed, if A[0..n − 1] is a strictly increasing array and we use A[0] as the pivot, the left-to-right scan will stop on A[1] while the right-to-left scan will go all the way to reach A[0], indicating the split at position 0: So, after making n + 1 comparisons to get to this partition and exchanging the pivot A[0] with itself, the algorithm will be left with the strictly increasing array A[1..n − 1] to sort. This sorting of strictly increasing arrays of diminishing sizes will continue until the last one A[n − 2..n − 1] has been processed. The total number of key comparisons made will be equal to $Cworst(n) = (n+1) + n + ... + 3 = (n+1)(n+2) - 3 \in \Theta(n2)$

```
// The quicksort algorthm works on the assumption that we can pick a pivot, and sort
// recursively. Once a pivot is chosen, swap it with the last array item. Starting with
// two pointers p1 at the beginning and p2 at the end advance pointers until the left
// is at a number larger than the pivot and the right is at a number smaller than the
// pivot. Swap these, then proceed repeating this until the points pass each other.
// Put the pivot back into its original position. Everything smaller is now to the left
// and everything larger is to the right. Call this function recursively on the left and
// right halves of the array excluding the pivot.

125 int quick(int arr[], int left, int right){
126     if(left < right){
127         int s = partition(arr, left, right);
128         quick(arr, left, s-1);
129         quick(arr, s + 1, right);
130     }
131     return 1;
132 }
135 int partition(int arr[], int left, int right){
136     int p = arr[left];
137     int i = left;
138     int j = right + 1;
139     do{
140         do{
141             ++i;
142         }while(arr[i] < p);
143         do{
144             --j;
145         }while(arr[j] > p);
146         swap(arr, i, j);
147     }while(i < j);
148     swap(arr, i , j); //undo last swap
149     swap(arr, left, j);
150     return j;
151
```