TACT Boot Camp
Sample Problems for Algorithms

1   Resources

Following is a list of publicly available resources used in much of the content presented during the lectures and in the exercises of the next sections.

1. "A Gentle Introduction to Algorithm Cmplexity Analysis": `http://discrete.gr/complexity/`

2. Algorithm analysis and complexity: `http://cs.lmu.edu/~ray/notes/alganalysis/`

3. Algorithms course lectures: `http://www3.cs.stonybrook.edu/~algorith/video-lectures/`

4. Greedy algorithms: `https://en.wikipedia.org/wiki/Greedy_algorithm`

5. Sorting: `https://www.cpp.edu/~ftang/courses/CS241/notes/sorting.htm`

6. Sorting algorithms overview: `https://en.wikipedia.org/wiki/Sorting_algorithm`

7. Several exercises from Sections 3 and 4 were taken from the following `https://www.cs.auckland.ac.nz/courses/compsci220s1t/lectures/lecturenotes/GG-lectures/220exercises1.pdf`

2   Algorithm Design

For each of the following, write an algorithm in pseudocode or in your favorite programming language to perform the function described.

Exercise 2.1. *(Summing arrays) Write a function which takes as arguments two arrays of integers, both assumed to be the same length, and an integer specifying their lengths, and returns the first array modified so that each element is the sum of the corresponding elements of the two array (i.e.* $A[0] = A[0] + B[0]$, $A[1] = A[1] + B[1]$, *etc.).*

> **Example Solution:**

```
int arraySum(A, B, n) {
    for( int i = 0; i < n; i++ ) {
        A[i] = A[i] + B[i];
    }
    return A;
}
```

Exercise 2.2. *(Counting matches) Write a function which takes as arguments two arrays of integers, both assumed to be the same length, and an integer specifying their lengths, and returns the sum total of the number of times that elements of the first array occur in the second. E.g., if the input arrays were* $\{1, 4, 7, 2, 5\}$ *and* $\{5, 10, 4, 5, 8\}$*, the function would return 3.*

Example Solution:

```
int countingMatches(A, B, n) {
    int matches = 0;
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            if (A[i] == B[j]) {
                matches = matches + 1;
            }
        }
    }
    return matches;
}
```

Exercise 2.3. *(Linear search) Write a function which takes as arguments an array of integers, an integer specifying its length, and an integer value to be searched for in the array. The function should return* -1 *if the search value is not found in the array, otherwise it should return the integer specifying the first index that that value is found in the array (e.g. if the search value is* 7 *and the* 3*rd element of the array, that is* $A[2]$*, is* 7*, it should return* 2*).*

Example Solution:

```
int linearSearch(A, n, s) {
    for( int i = 0; i < n; i++ ) {
        if (A[i] == s) {
            return i;
        }
    return -1;
}
```

Exercise 2.4. *(Binary search) Write a function which takes as input an array of integers which can be assumed to be sorted in increasing order, as well as two integers specifying the beginning and end indices to search, and an integer specifying a value to be searched for. It should return the index in the array at which the search value is found (if it is found), and* -1 *if it is not. This function should use recursion, and if the first call to it is with start index value* 0 *and end index value* $n - 1$ *(where* n *is the length of the array), it should complete in time* $O(\log n)$*.*

Example Solution:

```
int binarySearch(A, begin, end, s) {
    if (A[begin] == s) {
        return begin;
    } else if (begin == end) {
        return -1;
    }

    int middle = begin + (end-begin)/2;
    if (A[middle] == s) {
        return middle;
    } else if (A[middle] > s) {
        return binarySearch(A, begin, middle, s);
```

```
        } else {
            return binarySearch(A, middle, end, s);
        }
    }
```

Exercise 2.5. *(Fibonacci numbers) Write a function which takes as an argument a non-negative integer n and returns the nth Fibonacci number. It should use recursion.*

The *Fibonacci series* is defined as follows:

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

Example Solution:

```
int fibonacci(n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

3 Big-O Notation

Exercise 3.1. *For each of the expressions below, select the dominant term having the steepest increase in n and specify the lowest Big-O complexity of each expression.*

Solution:

Expression	Dominant term(s)	$O(\ldots)$
$10n^2 + 20n + 100$	$10n^2$	$O(n^2)$
$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
$500n + 100n^{1.5} + 50n\log_{10} n$	$100n^{1.5}$	$O(n^{1.5})$
$0.3n + 5n^{1.5} + 2.5n^{1.75}$	$2.5n^{1.75}$	$O(n^{1.75})$
$n^2\log_2 n + n(\log_2 n)^2$	$n^2\log_2 n$	$O(n^2\log n)$
$n\log_3 n + n\log_2 n$	$n\log_3 n,\ n\log_2 n$	$O(n\log n)$
$3\log_8 n + \log_2\log_2\log_2 n$	$3\log_8 n$	$O(\log n)$
$100n + 0.01n^2$	$0.01n^2$	$O(n^2)$
$0.01n + 100n^2$	$100n^2$	$O(n^2)$
$2n + n^{0.5} + 0.5n^{1.25}$	$0.5n^{1.25}$	$O(n^{1.25})$
$0.01n\log_2 n + n(\log_2 n)^2$	$n(\log_2 n)^2$	$O(n(\log n)^2)$
$100n\log_3 n + n^3 + 100n$	n^3	$O(n^3)$
$0.003\log_4 n + \log_2\log_2 n$	$0.003\log_4 n$	$O(\log n)$

4 Time Complexity of Code

For each of the following pieces of code, determine the time complexity.

Exercise 4.1. *Determine the time complexity of the following piece of code:*

```
if (i < n) {
    ... // constant number of operations
    if (k > i) {
        ... // constant number of operations
    }
}
```

Solution: In the worst case, all code is executed, but since there are no loops, there are only a constant number of operations in total and the code executes in $O(1)$ constant time.

Exercise 4.2. *Determine the time complexity of the following piece of code:*

```
for ( int  i  =  n;   i  >  0;   i  =  i  /  2  )  {
    for ( int  j  =  1;   j  <  n;   j  =  j  *  2  )  {
        for ( int  k  =  0;   k  <  n;   k  =  k  +  2  )  {
            ...  //  constant  number  of  operations
        }
    }
}
```

Solution: In the outer for-loop, the variable i keeps halving so it goes round $\log_2 n$ times. For each i, the next loop goes round also $\log_2 n$ times, because of doubling the variable j. The innermost loop by k goes round $n/2$ times. Loops are nested, so the bounds may be multiplied to give that the algorithm is $O(n(\log n)^2)$.

Exercise 4.3. *Determine the time complexity of the following piece of code:*

```
for  (  i=1;  i  <  n;   i  =  i  *  2  )  {
    for  (  j  =  n;   j  >  0;   j  =  j  /  2  )  {
        for  (  k  =  j;   k  <  n;   k  =  k  +  2  )  {
            sum  +=  (i  +  j  *  k  );
        }
    }
}
```

Solution: Running time of the inner, middle, and outer loops are proportional to n, $\log n$, and $\log n$, respectively. Thus the overall complexity is $O(n(logn)^2)$.

Exercise 4.4. *Determine the time complexity of the following piece of code assuming that $n = 2^m$:*

```
for ( int  i  =  n;   i  >  0;   i  =  i  −  1  )  {
    for ( int  j  =  1;   j  <  n;   j  =  j  *  2  )  {
        for ( int  k  =  0;   k  <  j;   k++  )  {
            ...  //  constant  number  C  of  operations
        }
    }
}
```

Solution: The outer for-loop goes round n times. For each i, the next loop goes round $m = \log_2 n$ times, because of doubling the variable j. For each j, the innermost loop by k goes round j times, so that the two inner loops together go round $1 + 2 + 4 + \ldots + 2^{m1} = 2^m - 1 \approx n$ times. Loops are nested, so the bounds may be multiplied to give that the algorithm is $O(n^2)$.

Exercise 4.5. *Determine the time complexity of the following piece of code:*

```
for ( int  bound  =  1;   bound  <=  n;   bound  =  bound  *  2  )  {
    for ( int  i  =  0;   i  <  bound;   i++  )  {
        for ( int  j  =  0;   j  <  n;   j  =  j  +  2  )  {
            ...  //  constant  number  of  operations
```

```
        }
        for ( int  j  =  1;  j  <  n;  j  =  j  *  2  )  {
            ...  //  constant  number  of  operations
        }
    }
}
```

Solution: The first and second successive innermost loops have $O(n)$ and $O(\log n)$ complexity, respectively. Thus, the overall complexity of the innermost part is $O(n)$. The outermost and middle loops have complexity $O(\log n)$ and $O(n)$, so a straightforward (and valid) solution is that the overall complexity is $O(n^2 \log n)$.

Exercise 4.6. *Assume that the array a contains n values, that the method randomValue takes a constant number c of computational steps to produce each output value, and that the method goodSort takes $n \log n$ computational steps to sort the array. Determine the time complexity for the following fragments of code:*

```
for (  i  =  0;  i  <  n;  i++  )  {
    for (  j  =  0;  j  <  n;  j++  )  {
        a [ j ]  =  randomValue (  i  );
    }
    goodSort (  a  );
}
```

Solution: The inner loop has linear complexity cn, but the next called method is of higher complexity $n \log n$. Because the outer loop is linear in n, the overall complexity of this piece of code is $n^2 \log n$.

5   Greedy Algorithms

Recall that a *greedy* algorithm is one which, at each step while determining a solution, selects the move that appears to be the "best" (by some measure). It does this without regard for future consequences of that selection, in the hope that by choosing a local optimum at each step, a global optimum (or close enough) will be discovered. Greedy algorithms are typically used for problems for which it is too computationally difficult to calculate the globally (overall) best solutions.

Exercise 5.1. *Given the following list of durations for a set of tasks to be scheduled on a group of 3 computers, use a greedy algorithm to assign the tasks to the computers. The greedy algorithm should always assign the longest unassigned task to the computer which has had tasks summing to the smallest total time assigned to it.*

Tasks durations: 22, 21, 19, 18, 14, 13, 9, 4

Example Solution:

1. Computer 1: 22, 13, 9 (Total: 44)

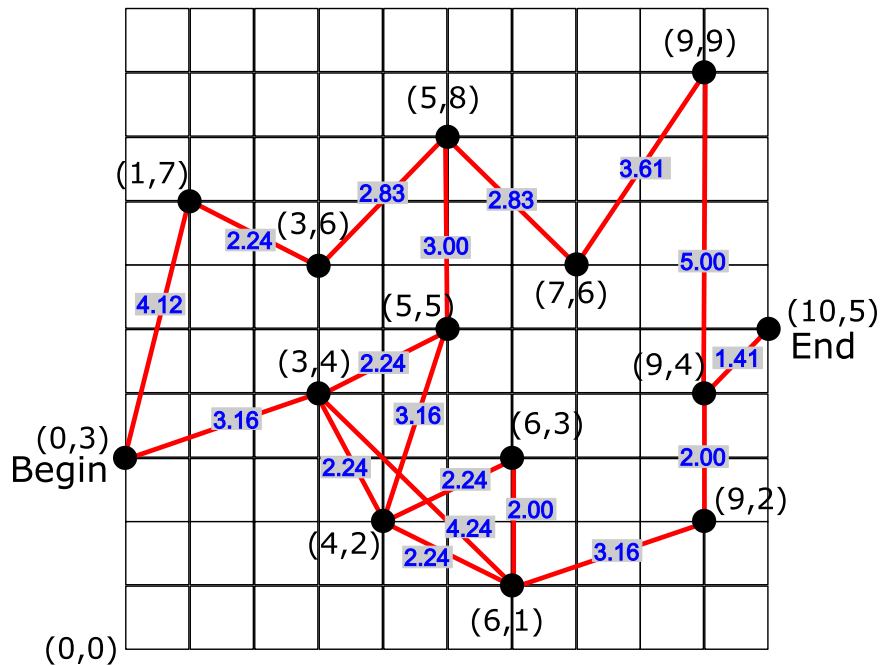2. Computer 2: 21, 14, 4 (Total: 39)

3. Computer 3: 19, 18 (Total: 37)

Figure 1: A map showing roads (red lines) between towns (black circles). The length of each road is given, along with the coordinates of each town.

Exercise 5.2. *Show that there is a better solution to Exercise 5.1 than the one found using the greedy algorithm.*

    Example Solution:

1. Computer 1: 22, 14, 4 (Total: 40)

2. Computer 2: 21, 19 (Total: 40)

3. Computer 3: 18, 13, 9 (Total: 40)

    This is a better solution because the overall running time is only 40 minutes, while the greedy algorithm results in a total time of 44 minutes.

Exercise 5.3. *Develop a greedy algorithm for the following task: Given a road map and a starting location, attempt to find a short(est) route to a given destination.*

    Example Solution: At each current position, select a road segment leading away from the current point which (1) has not been previously taken, and (2) leads to a point which is closest (as the crow flies) to the destination. If there are no such points, then backtrack by following the road which returns you closest to the starting location along the path(s) already taken, and continue.

Exercise 5.4. *Apply your greedy algorithm to the map in Figure 1 to attempt to find a short(est) path between the towns at $(0,3)$ and $(10,5)$ and its length.*

    Example Solution: Figure 2 shows the route found by the greedy algorithm presented as an example solution to Exercise 5.3 in blue (whose length is 21.25). (It also shows the optimal route, whose length is 13.97, in green.)
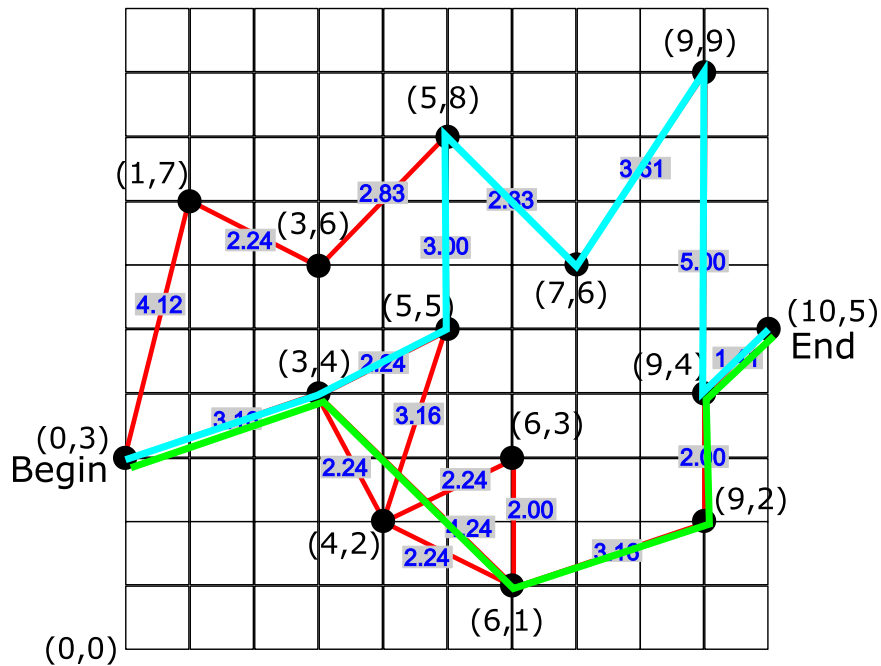
Figure 2: The map of Figure 1 showing the route found by the greedy algorithm of the example solution in blue (length = 21.25), and the optimal route in green (length = 13.97).

6 Sorting Algorithms

Exercise 6.1. *Give the average and worst case time complexities for each sorting algorithm, as well as the amount of additional memory required (beyond that needed for the input).*

Solution:

Name	Average	Worst	Memory
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quicksort	$O(n \log n)$	$O(n^2)$	$O(1)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(1)$

6.1 Bubble sort

The following is an implementation of bubble sort:

```
void bubbleSort(int array[], n) {
    for (int c = 0 ; c < ( n - 1 ); c++) {
        for (int d = 0 ; d < n - c - 1; d++) {
            if (array[d] > array[d+1]) /* For decreasing order use < */ {
                int swap    = array[d];
                array[d]    = array[d+1];
                array[d+1]  = swap;
            }
        }
    }
}
```

Exercise 6.2. *Demonstrate the changes made by each swap operation of bubble sort when performed on the following array:* | 7 | 5 | 4 | 2 | 3 |

Solution:

7	5	4	2	3
5	7	4	2	3
5	4	7	2	3
5	4	2	7	3
5	4	2	3	7
5	4	2	3	7
4	5	2	3	7
4	2	5	3	7
4	2	3	5	7
4	2	3	5	7
2	4	3	5	7
2	3	4	5	7
2	3	4	5	7
2	3	4	5	7

6.2 Selection sort

The following is an implementation of selection sort:

```
void selectSort(int arr[], int n) {
    //pos_min is short for position of min
    int pos_min, temp;

    for (int i=0; i < n-1; i++) {
        pos_min = i;   //set pos_min to the current index of array

            for (int j=i+1; j < n; j++) {

            if (arr[j] < arr[pos_min]) {
                pos_min = j;
                // pos_min will keep track of the index that
```

```
                    // min is in, this is needed when a swap happens
            }
        }

        // if pos_min no longer equals i than a smaller value must
        // have been found, so a swap must occur
        if (pos_min != i) {
            temp = arr[i];
            arr[i] = arr[pos_min];
            arr[pos_min] = temp;
        }
    }
}
```

Exercise 6.3. *Demonstrate the minimum valued elements found during each iteration and the changes made by each swap operation of selection sort when performed on the following array:*

7	5	4	2	3

Solution:

7	5	4	2	3
2	5	4	7	3
2	5	4	7	3
2	3	4	7	5
2	3	4	7	5
2	3	4	7	5
2	3	4	5	7
2	3	4	5	7

6.3 Insertion sort

The following is an implementation of insertion sort:

```
void insertion_sort (int arr[], int length) {
    int j, temp;

    for (int i = 1; i < length; i++) {
        j = i;

        while (j > 0 && arr[j] < arr[j-1]){
            temp = arr[j];
            arr[j] = arr[j-1];
            arr[j-1] = temp;
            j--;
        }
    }
}
```

Exercise 6.4. *Demonstrate the changes made by each swap operation of insertion sort when per-formed on the following array:* | 7 | 5 | 4 | 2 | 3 |

Solution:

7	5	4	2	3
5	7	4	2	3
5	7	4	2	3
5	4	7	2	3
5	4	7	2	3
4	5	7	2	3
4	5	7	2	3
4	5	2	7	3
4	5	2	7	3
4	2	5	7	3
4	2	5	7	3
2	4	5	7	3
2	4	5	7	3
2	4	5	3	7
2	4	5	3	7
2	4	3	5	7
2	4	3	5	7
2	3	4	5	7

6.4 Merge sort

The following is high-level pseudocode for an implementation of merge sort:

```
MergeSort(arr[], l,  r)
    If r > l
      1. Find the middle point to divide the array into two halves:
            middle m = (l+r)/2
      2. Call mergeSort for first half:
            Call mergeSort(arr, l, m)
      3. Call mergeSort for second half:
            Call mergeSort(arr, m+1, r)
      4. Merge the two halves sorted in step 2 and 3:
            Call merge(arr, l, m,  r)
```

Exercise 6.5. *Demonstrate how the array is first split apart, and then how it is merged when merge sort when performed on the following array:* | 8 | 5 | 7 | 2 | 6 | 1 | 4 | 9 | 3 |

Solution: (See Figure 3)

```
8 5 7 2 6 1 4 9 3
8 5 7 2      6 1 4 9 3
8 5   7 2    6 1    4 9 3
8  5  7  2   6  1   4  9 3
                       9  3
```

(a)

```
1 2 3 4 5 6 7 8 9
2 5 7 8      1 3 4 6 9
5 8   2 7    1 6    3 4 9
8  5  7  2   6  1   4  3 9
                       9  3
```
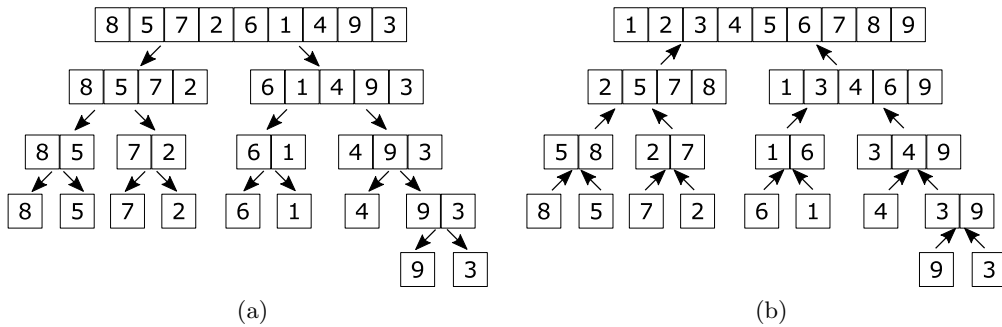
(b)

Figure 3: (a) Splitting the array in merge sort, (b) Merging the subarrays back together

6.5 Quicksort

The following is an implementation of quicksort:

```c
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
int partition (int arr[], int low, int high) {
    int pivot = arr[high];     // pivot
    int i = (low - 1);   // Index of smaller element

    for (int j = low; j <= high- 1; j++) {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;      // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
 arr[] --> Array to be sorted,
  low   --> Starting index,
  high   --> Ending index */
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        /* pi is partitioning index, arr[p] is now
           at right place */
        int pi = partition(arr, low, high);
```

```
            // Separately  sort  elements  before
            // partition  and  after  partition
            quickSort(arr, low, pi − 1);
            quickSort(arr, pi + 1, high);
        }
    }
```

Exercise 6.6. *Demonstrate how the array is sorted by showing which elements are selected as pivot elements, how the subarrays are sorted with respect to those pivot elements, and how the process is applied again to each subarray when quick sort when performed on the following array:*

| 8 | 5 | 7 | 2 | 6 | 1 | 4 | 9 | 3 |

Solution: (See Figure 4)
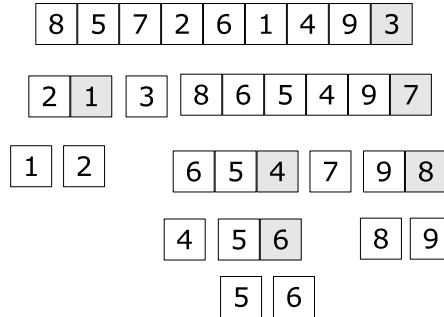
Figure 4: From top to bottom, how the pivot elements (in grey) are selected, then the elements less (greater) than that are placed in the left (right) subarray, then the process is repeated on each half.

6.6 Heapsort

The following is high-level pseudocode for an implementation of heapsort:

```
heapify  the  array;
while  the  array  is  not  empty {
    remove  and  replace  the  root;
    reheap  the  new  root  node;
}
```

We will first determine how to heapify the array | 8 | 5 | 7 | 2 | 6 | 1 | 4 | 9 | 3 |, and then we will demonstrate the process for removing and `reheap`ing.

6.6.1 Heapify

High-level pseudocode for `heapify`:

```
for  each  element  in  the  array {
    add  the  element  as  a  new  node  in  a  binary  tree:
        if  the  deepest  level  is  full: as  the  leftmost  leaf  of  a  new  level
        else: immediately  right  of  the  rightmost  leaf  of  the  bottom  level
    currNode = newly  added  leaf
```

```
        while currNode value is greater than its parent value {
            siftUp (to siftUp, we swap the values of currNode and its parent)
            currNode = parent
        }
    }
```

Exercise 6.7. *Demonstrate how* `heapify` *is performed on the following array:* | 8 | 5 | 7 | 2 | 6 | 1 | 4 | 9 | 3 |
Figure 5 shows the first 5 *node additions of* `heapify`, *until a call to* `siftUp` *if required. Demonstrate the results of that, as well as the rest of the node additions and any necessary* `siftUp` *operations along the way.*
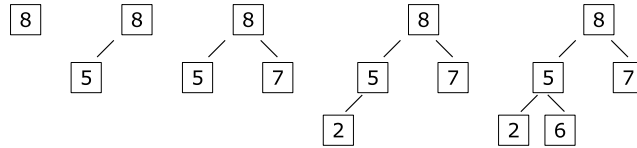
Figure 5: From left to right, the first 5 node additions when building a heap from the given array, but before `siftUp` is performed.
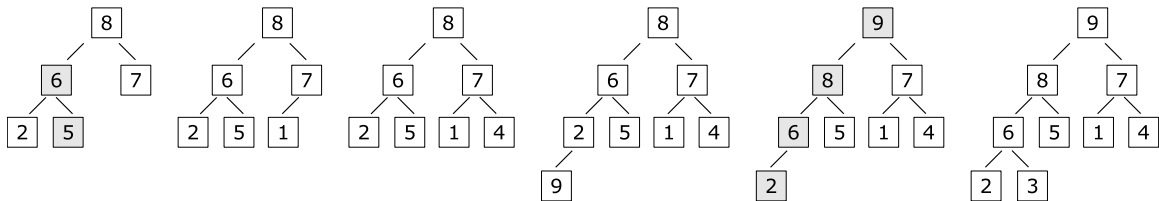
Solution: (See Figure 6)

Figure 6: Completion of `heapify` on the given array. From left to right, the result of applying `siftUp` (nodes in grey had values swapped) and remaining node additions and additional necessary calls to `siftUp`

6.6.2   Remove nodes and reheap

To complete the sorting of the array after forming a heap we perform the following:

```
    while the array is not empty {
        remove and replace the root;
        reheap(root);
    }
```

Recall that removing a node always consists of removing the root, and then we replace it with the rightmost leaf of the bottom level. The `reheap` operation is essentially the opposite of `siftUp`:

```
    reheap(currNode) {
        while value of currNode is less than one of its children {
            swap the values of currNode and its largest child
            currNode = node of swapped child
```

14

```
            }
      }
```

Exercise 6.8. *Figure 7 shows the initial heap before any node removals, and then the removal of the root. Demonstrate how that node is replaced, and then all necessary* `reheap` *operations to produce the new heap.*
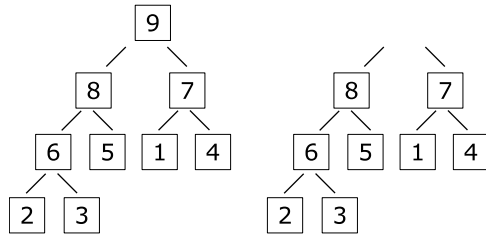
Figure 7: Initial heap, then with root removed
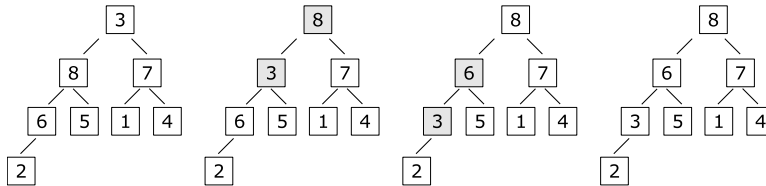
Solution: (See Figure 8)

Figure 8: Replacement of root node, followed by all necessary `reheap` operations