1. The sum of the powers of two (2^0 + 2^1 + ⋯ + 2^n) is equal to -----**2^(n+1)-1**------

2. The sum of increasing integers (1 + 2 + 3 + + n) is equal to --- **n*(n+1)/2**----

3. A logarithmic algorithm has a time complexity of O(**log n**)

4. A quadratic algorithm has a time complexity of O(**n^2**)

5. A linear algorithm has a time complexity of O(**n**)

6. A binary tree that has 10,000 internal nodes will have a height between ---**15**-- and ---**(10000+1)**--

7. A red-black tree that has 10,000 internal nodes will have a height between ---**15**-- and --**30**—

8. An AVL, tree that has 10,000 internal nodes will have a height between ---**15**-- and –**30**-

9. Pre-order traversal of a binary tree means the parent, left child, and right child nodes are "visited" in the following order --**parent**---, --**left**---, and --**right**---

10. Post-order traversal of a binary tree means the parent, left child, and right child nodes are "visited" in the following order ---**left**--, ---**right**--, and ---**parent**--

11. In-order traversal of a binary tree means the parent, left child, and right child nodes are visited in the following order -----, ----- and -----
    Answer: **(left, parent ,right)**
12. The maximum depth of any external node of a tree T is called the ---**Height**--- of T
13. In a **circular**, **growable** array implementation of the Queue ADT, the enqueue and dequeue operations run in ---**O(n)**-- and --**O(1)**-- amortized time if the array size is increased by a constant C each time it has to be enlarged
14. In a growable array-based implementation of a Stack ADT, the push operation runs in O(1), amortized worst-case time when the array size increases by ---**1**-- each time re-sizing is necessary

15. In a **hash table** implementation of the **Dictionary ADT**, the insertItem, findElement, and removeElement operations run in –**O(1)**---, --**O(1)**---, and –-**O(1)**--- expected time respectively.

16. In an **unsorted**, **growable array** implementation of the **Dictionary ADT**, the insertItem, findElement, and removeElement operations run in –-**O(1)**--, --**O(n)**--, --**O(n)**-- and time respectively.

17. In a **sorted**, **growable array** implementation of the **Dictionary ADT**, the insertItem, findElement, and removeElement operations run in --**O(n)**--, -- **O(log n)**--, and --**O(n)**-- time respectively

18. In a **red-black tree** implementation of the **Dictionary ADT**, the insertItem, findElement, and removeElement operations run in -- **O(log n)**---, -- **O(log n)**--, and --**O(log n)**-- time respectively

19. To sort a large sequence of keys that can-not fit in memory, -- **Merge Sort**--- is the recommended choice

20. In a real-time scenario where a sort must be completed within a **fixed amount of time** and the input **can fit in memory**, then -- **Heap sort** -- is the recommended choice

21. To sort a sequence of less than fifty keys, --**Insertion---**, is recommended choice.

True/False

1. The standard **Bucket-sort** distributes the keys into buckets, sorts each bucket, then concatenates the buckets. This algorithm runs in **linear time** no matter how the keys distributed over the buckets / **FALSE**

2. A sorting algorithm is considered **in-place** if it uses some memory in addition to the memory used by the input sequence, but only a constant amount, i.e., not an amount that increases as n increases / **TRUE**

3. In a **self-balancing binary search tree**, to remove a key-element pair, the **removeElement(k)** method calls **expandExternal(v) / FALSE**

4. In a binary search tree, every internal node has either one or two children / **FALSE**

5. The **lower bound on sorting by key** comparison is O(n) since we can always do a bucket or radix sort / **FALSE correct: O(n log n)**

6. All implementation of an **unordered dictionary** are necessarily inefficient for finding items since the entire dictionary might have to be scanned to find the key **/FALSE**

7. In **Radix**-sort, the key is divided into **components** and **Bucket-sort** is run on each component, starting with the most-significant (high order) component down to the least significant component (low order). / **FALSE**

8. In a Red-Black tree, the restructuring and recoloring operations are sometimes necessary when searching the tree. **/FALSE**

9. **Quick-sort** is an example of the **divide-and-conquer** approach with worst-case time complexity that is **no better** than **Selection-sort or Insertion-sort**. **/TRUE**

10. In a **heap**, external nodes **cannot appear on more than one level. / FALSE**

You are given an algorithm A with running time (n")) in every case (best case, worst case, and average case) and algorithm B with running time O(n) in every case. Very briefly describe why A might be faster than B on some inputs. Your answer cannot have anything to do with memory limitations, disk accesses, or paging (i.e., memory is unbounded, disk accesses take 1 unit of time, etc.)

(a) [15 points] Given a sequence of all books in a library (containing title, author, call number, and publisher) and another Sequence of 30 publishers, design an efficient algorithm to determine how many of the books were published by each publisher.
(b) [5 points] What is the time complexity of your algorithm? Justify your answer.

(a) [20 points] Given a Sequence B of thousands of credit card bills and another Sequence P of thousands of payments, design an efficient algorithm to create a Sequence of credit card bills that were not paid in full. The elements of Sequence B contain the credit card number, amount due, name, and address. The elements of Sequence P contain the credit card number, amount paid, and name. The output of the algorithm should be a newly created Sequence containing the unpaid bills; i.e., elements with the credit card number, name, address, amount due, and amount paid for those customers for which the amount paid is less than the amount due. Note that you must handle the case where there is a bill, but there is no payment.
(b) [5 points] What is the time complexity of your algorithm? Justify your answer

[20 points] Design an efficient algorithm to calculate the height and the balance factor of each internal node of a binary tree. The balance factor of an internal node is the height of the left subtree minus the height of the right subtree. Use the methods of the Tree and Binary Tree ADTs to traverse the tree. To set the height and balance factor of each internal node, use methods setHeight(v, h) and setBalFactor(v, bf) where v is a node of the tree, h is the height calculated for node v, and bf is the balance factor of v. For example, in the Red-Black tree of Figure below, the height of the node containing 13 is 1 and the balance factor is 0. Similarly, the height of the node containing 10 is 3 and the balance factor is 0 whereas, the height of the node containing 15 is 4 and the balance factor is +1 (since the height of its left subtree is one more than the height of its right subtree, i.e., 3-2=1)

(b) [5 points] What is the time complexity of your algorithm if the tree is a red-black tree? What if the tree is neither a red-black tree nor an AVL tree? Justify your answers.

[5 points] Draw the corresponding 2-4 tree for the red-black tree in Figure 1.

(10 points] For the red-black tree in Figure 1, insert the keys 8, 7, and 5 (in this order) and redraw the tree after any necessary re-coloring and rebalancing. Clearly label the red nodes with R. Show the tree after key 8 is inserted, another tree after 7 is inserted, and a final tree after 5 is inserted (if you show other intermediate steps, then clearly label these three trees).

[5 points] Let A be an array of n integers. What is the time complexity of the following algorithm? Give your answer as a summation that specifies the precise number of times that the statement A[i] <- A[i] + A[j] + A[k] is executed. for i<- 0 to n-1 do   for j <- 0 to i do   for k <- 0 to i do   A[i] <- A[i] + A[j] + A[k]

1.2.

A **pre-order** traversal of a binary tree means the parent, left child, and right child nodes are "visited" in the following order:

- ◯ A. parent, left, right
- ◯ B. left, right, parent
- ◯ C. parent, right, left
- ◯ D. left, parent, right

2.

A sorting algorithm is by definition **in-place** if it uses no more than O(n) memory in addition to the memory used by the input sequence, i.e., memory increases by no more than O(n) when the input size is O(n) as in sorting with an auxiliary Priority Queue.

- ◯ True
- ◯ False

Reset Selection

3.

In a **real-time scenario** where a sort must be completed within a fixed amount of time (no worse than O(n log n)), we have O(1) random access, we want no wasted space (in-place), and the input can fit in memory, then the recommended choice is:

- A. Insertion Sort
- B. Heap Sort
- C. Merge Sort
- D. Quick Sort
- E. Radix Sort
- F. Priority Queue Sort

4.

In a **post-order** traversal of a binary tree means the parent, left child, and right child nodes are "visited" in the following order:

- A. right, left, parent
- B. left, parent, right
- C. parent, left, right
- D. left, right, parent

Reset Selection

5.

In a (**proper**) binary tree, every Position has exactly zero or two children (in a proper binary tree leaf/external nodes are considered nodes even if they don't contain data and could be implemented as a null reference).

- True
- False

Reset Selection

6.

Why was the array-based implementation of the **Sequence** ADT implemented as a **circular array**? You must select all that apply, but none that do not apply.

- [ ] A. To reduce running time of S.insertFirst(e) from O(n) to O(1).
- [ ] B. To reduce running time of S.AtRank(r) from O(n) to O(1).
- [ ] C. To reduce running time of S.remove(S.first()) from O(n) to O(1).
- [ ] D. To reduce running time of insertAtRank(r, e) from O(n) to O(1).
- [ ] E. To reduce running time of removeAtRank(r) from O(n) to O(1).

7.

What are the advantages of **merge** sort? **Choose all that apply.**

- [ ] A. Simple, short, easy to understand (7 lines of code)
- [ ] B. Optimal (n log n)
- [ ] C. In-place
- [ ] D. Locality of reference (data stays in cache memory more than other sort algorithms)
- [ ] E. Fewest key comparisons of any sort algorithm
- [ ] F. Used when random access is inefficient
- [ ] G. Fastest if the keys are integers and the number of digits is known so key comparisons are unnecessary.

8.

In a **sorted, circular array** implementation of the **Priority Queue**, the insertItem, minKey, and removeMin operations run in ___________, ___________, and ___________ time respectively.

- ◯ A. insertItem=O(n), minKey=O(1), and removeMin=O(1)
- ◯ B. insertItem=O(n), minKey=O(n), and removeMin=O(n)
- ◯ C. insertItem=O(log n), minKey=O(1), and removeMin=O(log n)
- ◯ D. insertItem=O(1), minKey=O(n), and removeMin=O(n)
- ◯ E. insertItem=O(log n), minKey=O(1), and removeMin=O(1)

Reset Selection

9.

What are the advantages of **PriorityQueue** sort? **Choose all that apply.**

- [ ] A. Simple, short, easy to understand (7 lines of code)
- [ ] B. Optimal (n log n)
- [ ] C. In-place
- [ ] D. Locality of reference (data stays in cache memory more than other sort algorithms)
- [ ] E. Fewest key comparisons of any sort algorithm
- [ ] F. Used when random access is inefficient
- [ ] G.

Fastest if the keys are integers and the number of digits is known so key comparisons are unnecessary .

10.

In a Red-Black Tree implementation of an Ordered Dictionary with unique keys, the search operation never traverses both the left and right child of a node in the tree.

- ( ) True
- ( ) False

Reset Selection

11.

What are issues related to proper implementation of a **hash table based** Dictionary?

☐ A.
The client/user of the hash table must implement a hashcode function that maps the key to an integer such that there are very few collisions with other hashcodes of keys.

☐ B.
The hash table must internally implement a hashcode function that maps the key to an integer such that there are few very collisions with other hashcodes of keys.

☐ C.
The client/user can create a compression function that results in few collisions based on the initial size of the table being a prime number.

☐ D.
The internal hash table implementation can create a compression function that results in fewer collisions by ensuring that the table size is a prime number at the beginning as well as when it is resized.

☐ E. The client/user must handle collisions through either chaining or probing.

☐ F. The internal implementation of the hash table must handle collisions through either chaining or probing.

☐ G. The client/user must make sure the table is resized when the load factor goes above .75.

☐ H.
The internal implementation must make sure the table is resized whenever the load factor goes above .75.

12.

In a **sorted array** implementation of the **LookUpTable** (Dictionary) ADT, the insertItem, findElement, and removeElement operations run in ___________, ___________, and ___________ time respectively.

◯ A. insertItem=O(log n), findElement=O(1), and removeElement=O(log n)

◯ B. insertItem=O(n), findElement=O(log n), and removeElement=O(n)

◯ C. insertItem=O(n), findElement=O(1), and removeElement=O(n)

◯ D. insertItem=O(log n), findElement=O(log n), and removeElement=O(log n)

◯ E. insertItem=O(log n), findElement=O(n), and removeElement=O(log n)

Reset Selection

13.

What is the minimum and maximum height of a **Red-Black tree** with **18,000** internal nodes? **Hint:** $2^{14} = 16{,}384$ **and** $2^{15} = 32{,}768$ **and** $2^{16} = 65{,}536$

- A. between 16 but no less and 22,000 but no more.
- B. between 15 but no less and 22,000 but no more.
- C. between 15 but no less and 28 but no more.
- D. between 16 but no less and 44,001 but no more.
- E. between 15 but no less and 30 but no more.
- F. between 15 but no less and 44,001 but no more.
- G. between 14 but no less and 28 but no more.

14.

The recommended choice to sort a large set of keys that **cannot fit in memory** is ________________.

- A. Selection Sort
- B. Insertion Sort
- C. Heap Sort
- D. Merge Sort
- E. Quick Sort
- F. Radix Sort
- G. Priority Queue Sort

Reset Selection

15.

What are the advantages of **Radix** sort? **Choose all that apply.**

- [ ] A. Simple, short, easy to understand (7 lines of code)
- [ ] B. Optimal (n log n) or better
- [ ] C. In-place
- [ ] D. Locality of reference (data stays in cache memory more than other sort algorithms)
- [ ] E. Fewest key comparisons of any sort algorithm
- [ ] F. Used when random access is inefficient
- [ ] G. Fastest if the keys are integers and the number of digits is known so key comparisons are unnecessary

16.

In a **heap-based** implementation of the **Priority Queue**, the insertItem, minKey, and removeMin operations run in __________, __________, and __________ time respectively.

◯ A. insertItem=O(n), minKey=O(n), and removeMin=O(n)

◯ B. insertItem=O(log n), minKey=O(log n), and removeMin=O(log n)

◯ C. insertItem=O(1), minKey=O(1), and removeMin=O(1)

◯ D. insertItem=O(log n), minKey=O(1), and removeMin=O(log n)

◯ E. insertItem=O(log n), minKey=O(1), and removeMin=O(1)

17.

To remove a key-element item from a self-balancing binary search tree (Red-Black or AVL), the removeElement(k) method always removes the node containing the key k.

◯ True

◯ False

Reset Selection

18.

In a Red-Black tree, the restructuring operations can be called up to O(log n) times during either the insertion or deletion operations.

◯ True

◯ False

Reset Selection

19.

What are the advantages of **heap** sort? **Choose all that apply.**

- [ ] A. Simple, short, easy to understand (7 lines of code)
- [ ] B. Optimal (n log n)
- [ ] C. In-place
- [ ] D. Locality of reference (data stays in cache memory more than other sort algorithms)
- [ ] E. Fewest key comparisons of any sort algorithm
- [ ] F. Used when random access is inefficient
- [ ] G. Fastest if the keys are integers and the number of digits is known so key comparisons are unnecessary

20.

To sort an array-based sequence of **less than fifty keys**, the recommended choice is:

○ A. Insertion Sort

○ B. Merge Sort

○ C. Radix Sort

○ D. Heap Sort

○ E. Priority Queue Sort

○ F. Selection Sort

○ G. Quick Sort

21.

What is the minimum and maximum height of a **generic binary tree** with **22,000** internal nodes? **Hint: 2^{14} = 16,384 and 2^{15} = 32,768 and 2^{16} = 65,536**

○ A. height can be between 15 but no less and 22,000 but no more.

○ B. height can be between 15 but no less and 44,001 but no more.

○ C. height can be between 15 but no less and 16 but no more.

○ D. height can be between 14 but no less and 15 but no more.

○ E. height can be between 14 but no less and 28 but no more.

○ F. height can be between 16 but no less and 22,000 but no more.

○ G. height can be between 22,000 but no less and 44,001 but no more.

3. ___**T**___Generally, an algorithm that runs in O(n log n) time will take longer than an algorithm that has O(log n) time complexity when n > n_0.

4. ___**F**__When deciding between a List and an Array data structure, if the application will be frequently accessing the elements by rank and seldom inserting elements by rank then it is better to choose a List to store the elements.

5. __**F**___In the Queue ADT, the enqueue and dequeue operations run in O(log n) time.

6. **FALSE :** Post-order traversal of a tree means the node is “visited” after the node's parent is “visited”.

7. An algorithm with O(n^2) average case time complexity that takes 10 seconds to execute for an input size of 1000 elements will take how long to run when the input size is 10,000 elements.
a) less than 50 seconds
b) from 50 up to 500 seconds
c) from 500 up to 5000 seconds
d) from 5000 up to 50,000 seconds
e) more than 50,000 seconds

• Sequence ADT: first(), last(), before(p), after(p), replaceElement(p, o), swapElements(p, q), insertBefore(p,o), insertAfter(p, o), insertFirst(o), insertLast(o), remove(p), size(), isEmpty(), elemAtRank(r), replaceAtRank(r, o), insertAtRank(r, o), removeAtRank(r), atRank(r), rankOf(p)

• Dictionary ADT: findElement(k), insertltem(k, e), removeElement(k), items(), keys(), elements()

• OrderedDictionary: findElement(k), insertItem(k, e), removeElement(k), items(), closestKeyBefore(k),
closestKeyAfter(k), closestElemBefore(k), closestElemAfter(k)

• PriorityQueue: removeMin(), minKey(), minElement(k), insertltem(k, e)