Week 1

1. Is insertion sort stable? Justify your answer.
2. Use the big-O notation to represent the following function: $0.0001n^{1.5} + 0.000000005\sqrt{n^3}\log(10n) - 10000000\,n$.
3. Use the big-O notation to represent the following function: $\frac{0.01\,n^3}{10000\log(50\,n^{100})} + 10^{1000}\,n^2 + e^{0.0001\,n}$.
4. Use the big-O notation and exponential function to represent $n!$.
5. Given a sorted list in ascending order, can you convert it in descending order in $O(n)$ time?

1. Is insertion sort stable? Justify your answer.
2. Use the big-O notation to represent the following function: .
3. Use the big-O notation to represent the following function: .
4. Use the big-O notation and exponential function to represent .
5. Given a sorted list in ascending order, can you convert it in descending order in time?

1. Insertion sort is stable. This is because the current item is inserted on the right in the sorted list of the items before it. In particular, let $A[i] = A[j]$ and $i < j$. To avoid notation confustion let's assume that we create a new list B as the sorted list. Assume that after inserting $A[i]$ and right before $A[j]$ is inserted, we have $B[m] = A[i]$. The insertion algorithm finds k such that $B[k] \leq A[j] < B[k+1]$ and insert $A[j]$ at $B[k+1]$. Thus, $k \geq m$ and so $n = k+1 > m$.
2. $O(n^{1.5}\log n)$.
3. $O(e^{0.0001n})$.
4. Rough upper bound: $O(n^n)$. Tight upper bound using Stirling's formula: $O((n/e)^{n+1/2})$
5. Yes, it can be done. The simplest way to do so is to start from the right side of the sorted list and insert it to a new list from the left, one item at a time. However, this algorithm makes the new sorted list unstable. To fix this problem, let's first look at the following attempt: before we insert an item into the new list, we need to check if the number on the left of the item is equal to it, if not, insert it as stated in the algorithm. If yes, then keep comparing items on the left until the first that is not equal to the current item. Copy the entire segment of the items from left to right to the new list also from left to right to maintain stability. This attempt, however, incurs quadratic time in the worst case: Consider the sorted list of identical numbers. How can we obtain a stable sorted list in linear time? We modify the algorithm as follows: Start from the right side of the sorted list. Keep moving to its left until the first number on the left is different or exceeds the left boundary. Copy this segment into the new list in its original order.

Week 2

1. Recursive calls would be natural when implementing divide-and-conquer algorithms, but recursion incurs overheads. Are recusive calls the only way to implement divide-and-conquer algorithms?
2. Let n be a positive integer. How can you find the smallest integer n' greater than or equal to n such that n' is a power of 2?
3. Can you implement merge sort without using recursive calls? If so, describe your approach. If not, justify your answer.
4. Describe how you'd analyze the time complexity of a divide-and-conquer algorithm without using theh master's method.
5. Given the following recurrence relation T(n) = 2T(n//2) + n, where n//2 is the floor of n divided by 2 (Python operator), point out what is wrong in the following proof for showing that T(n) = O(n): T(n) = 2T(n//2) + n = 2O(n/2) + n = O(n) + n = O(n).
6. Why is an algorithm of time complexty $\Theta(n^{\ln 7})$ is asymptotically faster than an algorithm of time complexity $\Theta(n^3)$?

1.Recursive calls would be natural when implementing divide-and-conquer algorithms, but recursion incurs overheads. Are recusive calls the only way to implement divide-and-conquer algorithms?

2.Let n be a positive integer. How can you find the smallest integer n' greater than or equal to n such that n' is a power of 2?

3.Can you implement merge sort without using recursive calls? If so, describe your approach. If not, justify your answer.

4.Describe how you'd analyze the time complexity of a divide-and-conquer algorithm without using theh master's method.

5.Given the following recurrence relation T(n) = 2T(n//2) + n, where n//2 is the floor of n divided by 2 (Python operator), point out what is wrong in the following proof for showing that T(n) = O(n): T(n) = 2T(n//2) + n = 2O(n/2) + n = O(n) + n = O(n).

6.Why is an algorithm of time complexty $\Theta(n\ln 7)$ is asymptotically faster than an algorithm of time complexity $\Theta(n3)$?

1. Recursive calls would be natural when implementing divide-and-conquer algorithms, but recursion incurs overheads. Are recusive calls the only way to implement divide-and-conquer algorithms?
   - No. We can use a bottom up approach and use a table to store intermediate results from bottom up until the solution to the original problem is obtained.
2. Let n be a positive integer. How can you find the smallest integer n' greater than or equal to n such that n' a power of 2?
   - Let $n' = 2^{\text{ceil}(\log n)}$.
3. Can you implement merge sort without using recursive calls? If so, describe your approach. If not, justify your answer.
   - Yes. Assuming sorting in ascending order. Let A be the list of numbers to be sorted. Assume that len(A) = n. If n is not a power of 2, then pad A to produce B by filling in k extra cells on the right of A with values of M + 1, where M is the largest number in A, such that n+k is the smallest power of 2 greater than n. Now merge B[2i] with B[2i+1] to a new sublist B[2i..2i+1] which is sorted. Then merge B[2i..2i+1] with B[2i+2..2i+3] to a new sublist B[2i..2i+3] which is sorted. Keep merging newly generated newlists pairwise until there is only one sublist left. Then remove k numbers on the right from the result, which is A sorted.
4. Describe how you'd analyze the time complexity of a divide-and-conquer algorithm without using theh master's method.
   - First try to derive a solution to the underlying recurrence relation of time on the special case where the original size n can always be divisible according to the divide-and-conquer algorithm. Then for general inputs, use the solution to the special case as a guideline and try to prove that the solution also holds true for the general case using mathemtaical induction.
5. Given the following recurrence relation T(n) = 2T(n//2) + n, where n//2 is the floor of n divided by 2 (Python operator), point out what is wrong in the following proof for showing that T(n) = O(n): T(n) = 2T(n//2) + n = 2O(n/2) + n = O(n) + n = O(n).
   - This proof is wrong because what it's meant to show is that $T(n) \leq cn$ for all n with a fixed constant $c > 0$. But $T(n) = 2T(n//2) + n \leq 2c(n//2) + n \leq cn + n = (c+1)n$, not cn.
6. Why is an algorithm of time complexty $\Theta(n^{\ln 7})$ significantly faster than an algorithm of time complexity $\Theta(n^3)$?
   - This is because $\ln 7 < 3$ and so $n^{\ln 7}/n^3$ tends to 0 as n tends to infinity.

Week 3

1. In the greedy hiring strategy, why is the last person hired during the interviewing process the best candidate in the list?
2. Can you think of a better algorithm to help Alice find the best contractor with the same condition that Alice is facing?
3. Is it true that $n! \geq n^{n/2}$ for any positive number *n*. If yes, provide a rigorous mathematical proof If not, provide a counter example.
4. Let p be a large prime number and X_0 be a seed value. Write a Python program to compute $X_{n+1} = X_n^2 \bmod p$. Do the sequence of numbers generated this way look statistically random to you?
5. Search on the Internet and find out how to generate pseudorandom numbers in a given range of (a,b), where a < b.

1.In the greedy hiring strategy, why is the last person hired during the interviewing process the best candidate in the list?

2.Can you think of a better algorithm to help Alice find the best contractor with the same condition that Alice is facing?

3.Is it true that n factorial space greater or equal than n to the power of n divided by 2 end exponent for any positive number n. If yes, provide a rigorous mathematical proof If not, provide a counter example.

4.Let p be a large prime number and X0 be a seed value. Write a Python program to compute Xn+1 = Xn2 mod p. Do the sequence of numbers generated this way look statistically random to you?

5.Search on the Internet and find out how to generate pseudorandom numbers in a given range of (a,b), where a < b.

1. In the greedy hiring strategy, why is the last person hired during the interviewing process the best candidate in the list?
   - The last person hired is better than anyone hired before him/her according to the greedy hiring strategy. If this person is not the best, then it means someone else interviewed after this person is better and so that person will end up being hired. This contradicts to the assumption that this person is the last one hired.
2. Can you think of a better algorithm to help Alice find the best contractor with the same condition that Alice is facing?
   - Without any prior knowledge on the qualifications of the contractors in the list, what Alice can do is to rely on her own interviews. Since certain contractors may become unavailable after she interviews them but before she is done interviewing everyone in the list, the greedy strategy suggested by the algorithm firm is the best strategy. Other strategies such as random selection or first come first hired do not guarantee she can find the best contractor available at the time being interviewed.
3. Is it true that $n! \geq n^{n/2}$ for any positive number n. If yes, provide a rigorous mathematical proof If not, provide a counter example.
   - It is true. It can be done by showing that $k(n-k+1) \geq n$ for $1 \leq k \leq n//2$ using mathematical induction on k. Induction basis: $k = 1$. Trivial. Induction hypthesis: $k(n-k+1) \geq n$ for $1 \leq k < n//2$. Induction step: Want to show $(k+1)(n-(k+1)+1) \geq n$. We have $(k+1)(n-(k+1)+1) = (k+1)(n-k) = k(n-k+1) + n - 2k > k(n-k+1) \geq n$. This completes the industion step. We can show that for odd number n we have $(n//2 + 1)^2 \geq (n//2)^2 + 2(n//2) + 1 \geq n$ because $n//2 \geq n/2 - 1/2$. Hence, $(n!)^2 = \prod_{k=1}^{n} k(n-k+1) \geq n^n$. Hence, $n! \geq n^{n/2}$.
4. Let p be a large prime number and X_0 be a seed value. Write a Python program to compute $X_{n+1} = X_n^2$ mod p. Do the sequence of numbers generated this way look statistically random to you?
   - Yes. This simple square congruent method may be used as a PRNG.
   - ```
     p = 1009
     seed_number = int(input("Please enter a number:\n[____] "))
     number = seed_number
     already_seen = set()
     counter = 0

     while number not in already_seen:
         counter += 1
         already_seen.add(number)
         number = number * number % p
         print(f"#{counter}: {number}")

     print(f"We began with {seed_number} and"
           f" have repeated ourselves after {counter} steps"
           f" with {number}.")
     ```
5. Search on the Internet and find out how to generate pseudorandom numbers in a given range of (a,b), where a < b.
   - Return a random integer in (a,b): `random.randrange(a,b)`
   - Return a random float number in (a,b): `random.uniform(a,b)`

Week 4

1. What are the minimum and maximum numbers of elements in a heap of height *h*?
2. Show that an n-element heap has height (log n)//1 (i.e., the floor of log n with base 2).
3. Where in a max-heap might the smallest element reside, assuming that all elements are distinct?
4. Is the array with values [23; 17; 14; 6; 13; 10; 1; 5; 7; 12] a max-heap?
5. We can improve the running time of quicksort in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. Upon calling quicksort on a subarray with fewer than *k* elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. It can be shown that this sorting algorithm runs in O(*n* log *k* + *n* log (*n*/*k*)) expected time. How should we pick *k*, both in theory and in practice?
6. What is the tight lower bound of T(n) that satisfies T(n) = max {T(q) , T(n-q)} + cn for q in [1, n-1] and some postive constant *c*.

1. What are the minimum and maximum numbers of elements in a heap of height *h*?
2. Show that an n-element heap has height (log n)//1 (i.e., the floor of log n with base 2).
3. Where in a max-heap might the smallest element reside, assuming that all elements are distinct?
4. Is the array with values [23; 17; 14; 6; 13; 10; 1; 5; 7; 12] a max-heap?
     ```

5. We can improve the running time of quicksort in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. Upon calling quicksort on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. It can be shown that this sorting algorithm runs in $O(n \log k + n \log (n/k))$ expected time. How should we pick k, both in theory and in practice?

6. What is the tight lower bound of T(n) that satisfies $T(n) = \max \{T(q) , T(n-q)\} + cn$ for q in [1, n-1] and some postive constant c.

1. What are the minimum and maximum numbers of elements in a heap of height h?
   - The minimum number of elements is $2^0+2^1+2^2+\cdots+2^{h-1}+1=2^h-1+1=2^h$; the maximum number of elements is $2^0+2^1+2^2+\cdots+2^h=2^{h+1}-1$.
2. Show that an n-element heap has height $(\log n)//1$ (i.e., the floor of log n with base 2).
   - Let h be the height of the n-node heap, then it follows from item 1 we have $2^h \le n \le 2^{h+1}-1$. Hence, $h \le \log n < h+1$. Thus, $h = (\log n)//1$.
3. Where in a max-heap might the smallest element reside, assuming that all elements are distinct?
   - The smallest elelment in a max-heap must be at the bottom level.
4. Is the array with values [23; 17; 14; 6; 13; 10; 1; 5; 7; 12] a max-heap?
   - No, because 6 has children 5 and 7, and 7 is greater than 6.
5. We can improve the running time of quicksort in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. Upon calling quicksort on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. It can be shown that this sorting algorithm runs in $O(n \log k + n \log (n/k))$ expected time. How should we pick k, both in theory and in practice?
   - We want $k = n/k$. That is, $k = \sqrt{n}$.
6. What is the tight lower bound of $T(n)$ that satisfies $T(n) = \max_{1 \le q \le n-1} \{T(q) + T(n-q)\} + cn$ for some postive constant c.
   - Assuming q = 1 in all iterations and T(1) = 1, the we have T(n) >= cn(n+1)/2 + n. Hence, T(n) = $\Omega(n^2)$.

Week 5

1. Why is the process of comparison sorting of n numbers a binary decision tree with each leaf being a permution of the input?
2. Is counting sort stable? Justify your answer.
3. Can sorting be done without using comparisons at all?
4. Can linear-time sorting algorithms be in-place algorithms?
5. What happens if we perform radix sort by sorting the most significant digit first, then the next significant digit, and so on, to the least signifiant digit?

1.Why is the process of comparison sorting of n numbers a binary decision tree with each leaf being a permution of the input?

2.Is counting sort stable? Justify your answer.

3.Can sorting be done without using comparisons at all?

4.Can linear-time sorting algorithms be in-place algorithms?

5.What happens if we perform radix sort by sorting the most significant digit first, then the next significant digit, and so on, to the least signifiant digit?

1. Why is the process of comparison sorting of n numbers a binary decision tree with each leaf being a permution of the input?
    - Comparison sorting cares about only the ordering of the numbers, not the values. Since each comparison can be performed on just one pair, any sorting algorithm is a sequence of comparisons that leads to an output, where is the output is a permutation of the input. Since an input could be all possible ordering, the output should also cover all possible permutations, which is n!.
3. Is counting sort stable? Justify your answer.
    - Yes. Reason: If a value appears more than once, then the first appearance of the value from right to left is placed in the right position. Since the count is decreased, the next appearnce is also placed in the right position.
4. Can sorting be done without using comparisons at all?
    - Yes. Counting sort, radix sort, and bucket sort do not use comparisons.
5. Can linear-time sorting algorithms be in-place algorithms?
    - No, because they must rely on the values of the input numbers that require extra space to store information about the properties of these values.
6. What happens if we perform radix sort by sorting the most significant digit first, then the next significant digit, and so on, to the least signifiant digit?
    - It could violate the sorted order. E.g., for 32, 15, 41, we would have 15, 32, 41, then 41, 32, 15, which is not sorted.

Week 6

1. How to use kthSmallest(A, p, r, k) to find the kth largest element in A with n = len(A)?
2. Can you find the kth smallest item on n items with values in a small range compared to n in expected time of O(n)?
3. Can search in a BST of n nodes require n comparisons?
4. If we construct a BST on a sorted list in ascending order by reading the numbers from left to right, is this BST well balanced?
5. Let A be a list of 2n + 1 distinct numbers with A[0] being the median. What is the maximum height of the BST built on A from left to right?

1. How to use kthSmallest(A, p, r, k) to find the kth largest element in A with n = len(A)?
2. Can you find the kth smallest item on n items with values in a small range compared to n in expected time of O(n)?
3. Can search in a BST of n nodes require n comparisons?
4. If we construct a BST on a sorted list in ascending order by reading the numbers from left to right, is this BST well balanced?
5. Let A be a list of 2n + 1 distinct numbers with A[0] being the median. What is the maximum height of the BST built on A from left to right?

1. How to use kthSmallest(A, p, r, k) to find the kth largest element in A with n = len(A)?
    - kthSmallest(A, 0, n - 1, n - k + 1) is the kth largest in A.
2. Yes. We first use counting to remove duplicates in O(n) time. Let m be the total number of items left, which are distinct. Use kthSmallest to find the kth smallest number if it exists, which runs in expected O(m) time. Thus, the running time of this algorithm is expected O(n).
3. Can search in a BST of n nodes require n comparisons?
    - Yes. For example, if a BST is a linear structure with all children being on the right and the item x to be search is greater than or equal to the leaf node.
4. If we construct a BST on a sorted list in ascending order by reading the numbers from left to right, is this BST well balanced?
    - No. This BST is linear with all children being on the right.
5. Let A be a list of 2n + 1 distinct numbers with A[0] being the median. What is the maximum height of the BST built on A from left to right?
    - At most n, namely the longest path from the root to a leaf consists of at most n edges.

Week 7

1. If there are n nodes in an AVL tree, what is the minimum height of the AVL tree?
2. If there are n nodes in an AVL tree, what is the maximum height of the AVL tree?
3. If the height of an AVL tree is h, what is the maximum number of nodes can be in the tree?
4. Give a recurrence relation N(h) for the minimum number of nodes in a tree with height h.
5. What is the time complexity of searching, inserting, and deleting a node in an n-node AVL tree?

1. If there are n nodes in an AVL tree, what is the minimum height of the AVL tree?
2. If there are n nodes in an AVL tree, what is the maximum height of the AVL tree?
3. If the height of an AVL tree is h, what is the maximum number of nodes can be in the tree?
4. Give a recurrence relation N(h) for the minimum number of nodes in a tree with height h.
5. What is the time complexity of searching, inserting, and deleting a node in an n-node AVL tree?

1. If there are n nodes in an AVL tree, what is the minimum height of the AVL tree?
    - log n // 1 (the floor of log n).
2. If there are n nodes in an AVL tree, what is the maximum height of the AVL tree?
    - At most 1.44 log n.
3. If the height of an AVL tree is h, what is the maximum number of nodes can be in the tree?
    - $2^{h+1} - 1$.
4. Give a recurrence relation N(h) for the minimum number of nodes in a tree with height h.
    - N(h) = N(h−1) + N(h−2) + 1 for n > 2 where N(1) = 1 and N(2) = 2. This can be seen as follows: If we know N(h−1) and N(h−2), we can determine N(h). Note that this N(h)-noded tree must have a height h. Hence, the root of the tree must have a child that has height h − 1. To minimize the total number of nodes in this tree, we would have this sub-tree contain N(h−1) nodes. This implies that the minimum height of the other child is h − 2. By creating a tree with a root whose left sub-tree has N(h−1) nodes and whose right sub-tree has N(h−2) nodes, we have constructed the AVLtree of height h with the least nodes possible. This AVL tree has a total of N(h−1)+N(h−2) + 1 nodes (the 1 coming from the root itself).
5. What is the time complexity of searching, inserting, and deleting a node in an n-node AVL tree?
    - O(log n).

Week 8

1. Why is it true that in a red-black tree, the longest path is at most twice of the shortest path?
2. If we modify the definition a red-black tree to allow a red node to have a black child but not a black grandchild, where everything other property remains the same. What the longest path in terms of the shortest path.
3. Would it be easier to build a modified red-black tree described above and maintain its red-black properties?
4. Why is an AVL tree better than a red-black tree in terms of searching?
5. Which case of red-black property violation is the hardest to restore?

1. Why is it true that in a red-black tree, the longest path is at most twice of the shortest path?
2. If we modify the definition a red-black tree to allow a red node to have a black child but not a black grandchild, where everything other property remains the same. What the longest path in terms of the shortest path.

3. Would it be easier to build a modified red-black tree described above and maintain its red-black properties?
4. Why is an AVL tree better than a red-black tree in terms of searching?
5. Which case of red-black property violation is the hardest to restore?

1. Why is it true that in a red-black tree, the longest path is at most twice of the shortest path?
    - Since each path in a red-black tree must start and end with black nodes, and all paths must have the same number of black nodes, a path that contains no red nodes is the shortest while a path that contains red nodes alternatively, namely, black, red, black, red, ..., red, black, is the longest possible where the number of red nodes is the same as the number of black nodes minus 1. Let the number of blacks on a path be b + 1. Then the length of the shortest possible path is b and the lenght of the longest possible path is 2b. Thus, the longest path is at most twice of that of the shortet path.
2. If we modify the definition a red-black tree to allow a red node to have a black child but not a black grandchild, where everything other property remains the same. What the longest path in terms of the shortest path.
    - The longest possible path is in the form of black, red, red, black, ..., black, red, red, black, with two consecutive reds between two blacks. The shortest possible path is all black; let b + 1 be the number of blacks. Then the height of it is b. The number of nodes on the longest possible path is 3b + 1 and so the length is 3b. Thus, the longest path is at most thrice of that of the shortest path.
3. Would it be easier to build a modified red-black tree described above and maintain its red-black properties?
    - No, because essentially the same number of rotations would be needed.
4. Why is an AVL tree better than a red-black tree in terms of searching?
    - An AVL tree is a well balanced binary search tree while a red-black tree is balanced but not as well as an AVL tree on the same inputs. Thus, what takes h steps to search on an AVL tree may take upto 2h steps to search on a red-black tree.
5. An AVL tree requires an extra attribute of height for each node to help maintain balance. Do you think it would help to use an extra attribute of bh (black height) for each nodel in a red-black tree?
    - No. Red-black trees do not need this extra attribute to maintain the black-tree properties.

Week 9

1. A ***direct address table*** is the simplest hash table where keys are direct indexes of the table. Suppose that a dynamic set S is represented by a direct-address table T of length m and assume that there is no collision. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?
2. A ***bit vector*** is simply an array of bits (0s and 1s). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements with no satellite data. Dictionary operations should run in O(1) time.
3. Suppose we use a hash function h to hash n distinct keys into an array T of length m. Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of {{k,i}:k != i and h(k)=h(i)}?
4. Demonstrate what happens when we insert the keys 5,28,19,15,20,33,12,17,10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be h(k) = k mod 9.
5. Suggest how to allocate and deallocate storage for elements within the hash table itself by linking all unused slots into a free list, which is a list of unoccupied slots. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in O(1) expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?

1. A ***direct address table*** is the simplest hash table where keys are direct indexes of the table. Suppose that a dynamic set S is represented by a direct-address table T of length m and assume that there is no collision. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?
2. A ***bit vector*** is simply an array of bits (0s and 1s). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements with no satellite data. Dictionary operations should run in O(1) time.

3. Suppose we use a hash function h to hash n distinct keys into an array T of length m. Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of {{k,i}:k != i and h(k)=h(i)}?

4. Demonstrate what happens when we insert the keys 5,28,19,15,20,33,12,17,10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be h(k) = k mod 9.

5. Suggest how to allocate and deallocate storage for elements within the hash table itself by linking all unused slots into a free list, which is a list of unoccupied slots. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in O(1) expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?

1. A ***direct address table*** is the simplest hash table where keys are direct indexes of the table. Suppose that a dynamic set S is represented by a direct-address table T of length m and assume that there is no collision. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?
    - The maximum element of S could be in any slot and we don't know which slot holds the maximum number. Thus, we would need to search the entire hash table from slot to slot to find it, which is O(n) time in the worst case for a table of n items.

2. A ***bit vector*** is simply an array of bits (0s and 1s). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements with no satellite data. Dictionary operations should run in O(1) time.
    - We can represent keys of values from 0 to m - 1 using the bit vector data structure B[0..m - 1], where B[k] = 1 indicates slot k is not empty and 0 otherwise. For instance, we can store the set {0, 1, 2, 5 ,6,10,16, 19} in a bit vector of length 20 by 11100110001000001001. Dictionary operations are search for a key, insert a key, and delete a key. Since key k corresponds to the index in the bit vector B. To search for k, we simply check if B[k] = 1, if yes, return k, otherwise, return "not found". To insert k, simply set B[k] = 1. To delete k, simply set B[k] = 0. These operations all run in O(1) time.

3. Suppose we use a hash function h to hash n distinct keys into an array T of length m. Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of {{k,i}:k != i and h(k)=h(i)}?
    - Suppose that all the keys are totally ordered $\{k_1,\ldots,k_n\}$. Let X_i denote the number of ℓ's such that $\ell > k_i$ and $h(\ell)=h(k_i)$. So X_i is the (expected) number of times that key k_i is collided by those keys hashed afterward. Note that this is the same thing as $\sum_{j>i}\Pr(h(k_j) = h(k_i)) = \sum_{j>i}1/m = (n-i)/m$. Then, by linearity of expectation, the number of collisions is the sum of the number of collisions for each possible smallest element in the collision. The expected number of collisions is $\sum_{i=1,\ldots,n}(n-i)/m = n(n-1)/2m$.

4. Demonstrate what happens when we insert the keys 5,28,19,15,20,33,12,17,10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be h(k) = k mod 9.
    - Answer is given below:

$h(k)$	keys
$0 \mod 9$	
$1 \mod 9$	$10 \to 19 \to 28$
$2 \mod 9$	20
$3 \mod 9$	12
$4 \mod 9$	
$5 \mod 9$	5
$6 \mod 9$	$33 \to 15$
$7 \mod 9$	
$8 \mod 9$	17

5. Suggest how to allocate and deallocate storage for elements within the hash table itself by linking all unused slots into a free list, which is a list of unoccupied slots. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in O(1) expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?
    - The flag in each slot of the hash table T is 1 if the slot is occupied, and 0 if it is free. The free list must be doubly linked.
        - Search is unmodified, so it has expected time O(1).
        - To insert an element x, first check if T[h(x.key)] is free. If it is, delete T[h(x.key)] from the free list, insert it to T, and change the flag of T[h(x.key)] to 1. If it wasn't free to begin with, simply insert x.key at the start of the list stored there.
        - To delete, first check if x.prev and x.next are NIL. If they are, then the list will be empty upon deletion of x, so insert T[h(x.key)] into the free list, update the flag of T[h(x.key)] to 0, and delete x from the list it's stored in. Since deletion of an element from a singly linked list isn't O(1), we must use a doubly linked list.
        - All other operations are O(1).

Week 10

1. Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack is O(n) by assigning suitable amortized costs to the various stack operations.
2. If the set of stack operations included a MULTIPUSH operation, which pushes k items onto the stack, would the O(1) bound on the amortized cost of stack operations continue to hold?
3. Show that if a DECREMENT operation were included in the k-bit counter example, n operations could cost as much as Θ(nk) flips.
4. Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Assume that it takes O(1) time to examine or modify one bit. Figure out how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes time O(n) on an initially zero counter. (Hint: Introduce a new field C:max to hold the index of the high-order 1 in counter C. Initially, C:max is set to -1, since the low-order bit of A is at index 0, and there are initially no 1's in C. The value of C:max is updated as appropriate when the counter is incremented or reset, and we use this value to limit how much of C must be looked at to reset it. By controlling the cost of RESET in this way, we can limit it to an amount that can be covered by credit from earlier INCREMENTs.)
5. Show how to implement a queue with two ordinary stacks so that the amortized cost of each ENQUEUE and each DEQUEUE operation is O(1).
    -

1. Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack is O(n) by assigning suitable amortized costs to the various stack operations.
2. If the set of stack operations included a MULTIPUSH operation, which pushes k items onto the stack, would the O(1) bound on the amortized cost of stack operations continue to hold?
3. Show that if a DECREMENT operation were included in the k-bit counter example, n operations could cost as much as Θ(nk) flips.
4. Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Assume that it takes O(1) time to examine or modify one bit. Figure out how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes time O(n) on an initially zero counter. (Hint: Introduce a new field C:max to hold the index of the high-order 1 in counter C. Initially, C:max is set to -1, since the low-order bit of A is at index 0, and there are initially no 1's in C. The value of C:max is updated as appropriate when the counter is incremented or reset, and we use this value to limit how much of C must be looked at to reset it. By controlling the cost of

RESET in this way, we can limit it to an amount that can be covered by credit from earlier INCREMENTs.)

5. Show how to implement a queue with two ordinary stacks so that the amortized cost of each ENQUEUE and each DEQUEUE operation is O(1).

1. Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack is O(n) by assigning suitable amortized costs to the various stack operations.
    - Amortize cost for every push is two units, with one unit to pay for the push operation and the other to keep the item in the stack. Likewise, amortize cost for every pop is also two units, with one unit to pay for the pop operation and the other to copy the item. Thus, after every k operations, there are at least k units in the stack. Credit never goes negative, and the amortized complexity is O(n).
2. If the set of stack operations included a MULTIPUSH operation, which pushes k items onto the stack, would the O(1) bound on the amortized cost of stack operations continue to hold?
    - No.The time complexity of such a series of operations depends on the number of the MULTIPUSH operation. Since one MULTIPUSH needs Θ(k)time, performing n MULTIPUSH operations, each with k elements, would take Θ(kn)time, leading to amortized cost of Θ(k).
3. Show that if a DECREMENT operation were included in the k-bit counter example, n operations could cost as much as Θ(nk) flips.
    - In the worst case, going from 10^{k-1} to 01^{k-1} takes k flips (e.g., 10000 -> 01111 takes 5 flips). In the n operations, there could be any sequence of increment and decrement (e.g., 10000 -> 01111 -> 10000 -> 01111). When there are cn such operations for some positive constant c, the total number of flips.is Θ(n)k = Θ(nk).
4. Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Assume that it takes O(1) time to examine or modify one bit. Figure out how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes time O(n) on an initially zero counter. (Hint: Introduce a new field C:max to hold the index of the high-order 1 in counter C. Initially, C:max is set to -1, since the low-order bit of A is at index 0, and there are initially no 1's in C. The value of C:max is updated as appropriate when the counter is incremented or reset, and we use this value to limit how much of C must be looked at to reset it. By controlling the cost of RESET in this way, we can limit it to an amount that can be covered by credit from earlier INCREMENTs.)
    - We use the accounting method. Assume that it costs $1 to flip a bit. In addition, assume that it costs $1 to update C:max. Setting and resetting bits by INCREMENT will work exactly as for the original counter in the textbook: $1 will pay to set one bit to 1; $1 will be placed on the bit that is set to 1 as credit; the credit on each 1 bit will pay to reset the bit during incrementing. In addition, we'll use $1 to pay to update C:max. If C:max increases, we'll place an additional $1 of credit on the new high-order 1. (If C:max doesn't increase, we just don't use that $1.)

      Since RESET manipulates bits at positions only up to C:max, and since each bit up to there must have become the high-order 1 at some time before the high-order 1 got up to C:max, every bit seen by RESET has $1 of credit on it. So the zeroing of bits of C by RESET can be completely paid for by the credit stored on the bits. We just need $1 to pay for resetting C:max. Thus charging $4 for each INCREMENT and $1 for each RESET is sufficient, so the sequence of n INCREMENT and RESET operations takes O(n) time.

5. Show how to implement a queue with two ordinary stacks so that the amortized cost of each ENQUEUE and each DEQUEUE operation is O(1).
    - We use the first stack for ENQUEUE, and the second stack for DEQUEUE. In an ENQUEUE operation, we push the element into first stack. In a DEQUEUE operation, if second stack is not empty, pop the elements in the top of second stack. If the second stack is empty, pop all elements in first stack, and push them into second stack. Because this procedure will reverse the order of elements, we can directly pop elements in the top of second stack. Because all elements only push and pop only twice, the amortized cost for eace ENQUEUE and DEQUEUE operation is O(1).

Week 11

1. In general, what problems can be soloved using dynamic programming in polynomial time?
2. What are the general approach to figuring out a recurrence relation?
3. What is the top-down approach to solving a recurrence relation?
4. What is the bottom-up approach to solving a recurrence relation?
5. Dynamic programming is often used for solving optimization problems. Can it be used to solve non-optimization problems?

1. In general, what problems can be soloved using dynamic programming in polynomial time?
2. What are the general approach to figuring out a recurrence relation?
3. What is the top-down approach to solving a recurrence relation?
4. What is the bottom-up approach to solving a recurrence relation?
5. Dynamic programming is often used for solving optimization problems. Can it be used to solve non-optimization problems?

1. In general, what problems can be soloved using dynamic programming in polynomial time?
    - In general, any problem whose solution can be expressed as a recurrence relation where the current problem depends on the solution to at most polynomially many (in terms of the input length) subproblems of smaller sizes.
2. What are the general approach to figuring out a recurrence relation?
    - First, formulate the problem, specifying a name and parameters, that represent the solution to the problem. Next, determine how the current problem would depend on subproblems (this is localization).
3. What is the top-down approach to solving a recurrence relation?
    - Memoization: Store solutions to subproblems in a list or other types of data structure to cut down the cost of recurrences of old computations. You may still use recursion in coding.
4. What is the bottom-up approach to solving a recurrence relation?
    - Start from the smallest subproblems whose solutions are trivial (this is the stopping condition of the recurrence relation) and store the solutions. Then compute subproblems at the next level up based on the solutions to lower-level subproblems and store the solutions. Repeat until reaching the given parameters for the problem.
5. Dynamic programming is often used for solving optimization problems. Can it be used to solve non-optimization problems?
    - Yes. For example, Fibanacci numbers can be computed using dynamic programming efficiently.

Week 12

1. Why does memorization reduce computation time for solving a problem with a recurrence relation of subproblems?
2. Why does bottom-up approach reduce computation time for solving a problem with a recurrence relation of subproblems?
3. What are the pros and cons of top-down (memorization) and bottom-up (tabular) approaches?
4. How can you quickly determine the time complexity upper bound for a dynamic programming problem?
5. How do you determine if a problem can be solved using dynamic programming and what is the general approach of doing it?

1. Why does memorization reduce computation time for solving a problem with a recurrence relation of subproblems?
2. Why does bottom-up approach reduce computation time for solving a problem with a recurrence relation of subproblems?
3. What are the pros and cons of top-down (memorization) and bottom-up (tabular) approaches?
4. How can you quickly determine the time complexity upper bound for a dynamic programming problem?
5. How do you determine if a problem can be solved using dynamic programming and what is the general approach of doing it?

1. Why does memorization reduce computation time for solving a problem with a recurrence relation of subproblems?
    - Memorization avoids recomputing the same subproblem over and over again. Each subproblem may further depend on its subproblems, and the total number of depended subproblems may be exponential, yet only polynomially many subproblems are different that need to be computed. Thus, for each subproblem, we only want to compute it once.
2. Why does bottom-up approach reduce computation time for solving a problem with a recurrence relation of subproblems?
    - Botton-up approach does not use recursion. Instead, it directly starts computing the boundary subproblems (bottom), saves the results in a table, and then computes the subproblems one-level up, one level at a time, until the original problem is solved. This avoids recomputing the same subproblem over and over again.
3. What are the pros and cons of top-down (memorization) and bottom-up (tabular) approaches?
    - Memorization approach. Pros: It is easy to code because it is essentially the original recurrence relation. Cons: it incurs certain computation overhead because of recursive function calls.
    - Bottom-up approach. Pros: No overhad due to recursive function calls. Cons: It's a bit harder to code.
4. How can you quickly determine the time complexity upper bound for a dynamic programming problem?
    - Let k be the number of subproblems in the recurrence relation for the problem to be solved and n be the total number of different subproblems. Then the time complexity upper bound is O(kn) using either memorization or tabular method.
5. How do you determine if a problem can be solved using dynamic programming and what is the general approach of doing it?
    - First, determine if solving the original problem can be done by solving subproblems.
    - Second, come up with a recurrence relation by first naming the problem and parameters, and then specifying dependcies of subproblems.
    - Third, use either memorization or tabular method to obtain a solution.

Week 13

1. Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work.
2. Consider a modification to the activity-selection problem in which each activity aihas, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, we wish to choose a set A of compatible activities such that adding them up is maximized. Give a polynomial-time algorithm forthis problem.
3. Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.
4. Describe an efficient algorithm that, given a set $\{x_1, x_2, ..., x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.
5. What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers: a:1, b:1, c:2, d:3, e:5, f:8, g:13, h:21? Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

1. Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work.
2. Consider a modification to the activity-selection problem in which each activity aihas, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, we wish to choose a set A of compatible activities such that adding them up is maximized. Give a polynomial-time algorithm forthis problem.
3. Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

4. Describe an efficient algorithm that, given a set $\{x_1, x_2, ..., x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

5. What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers: a:1, b:1, c:2, d:3, e:5, f:8, g:13, h:21? Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

1. Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of the least duration from among those that are compatible with previously selected activities does not work.
    - Let a1 = [1, 5], a2 = [6, 10], a3 = [6, 10]. Use the greedy selection of the least duration, a3 will be selected, which prevents a1 and a2 from being selected becaue a3 overlaps with each of a1 and a2. However, a1 and a2 are compatible and should be selected instead.
2. Consider a modification to the activity-selection problem in which each activity aihas, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, we wish to choose a set A of compatible activities such that adding them up is maximized. Give a polynomial-time algorithm for this problem.
    - This problem can be solved in $O(n^3)$ time using DP as in the slides. In particular, let MaxA[i,j] denote the maximum solution for S_{ij}. Want to compute MaxA[1,n], where MaxA[i,j] = 0 if S_{ij} is empty and the maximum value of {MaxA[i,k] + MaxA[k,j] + v_k for a_k in A}.
3. Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.
    - When the order of increasing weight is the same as the order of decreasing value, the order of decreasing unit value is the same as the increasing weight. Thus, it suffices to select items with the smallest weight first, then items of next smallest weight, and so on. This algorithm takes O(n log n) time required for sorting. This algorithm is correct for the following reason: Let K be an optimal solution. If the greedy choice is not in K, then we can simply replace any item in K with the greedy choice to obtain a solution with a value at least the value of K. This shows that the greedy choice works.
4. Describe an efficient algorithm that, given a set $\{x_1, x_2, ..., x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.
    - Sort the numbers $x_1, x_2, ..., x_n$ in ascending order to $y_1 <= y_2 <= ... <= y_n$. Place $[y_1, y_1 + 1]$ into the solution set. Scan from left to right. For each y_i encountered, if it belongs to last interval in the solution set, remove it; otherwise, place $[y_i, y_i + 1]$ into the solution set. This algorithm takes O(n log n) time for sorting. Its correctness can be proven as follows: Suppose that the greedy choice is not in the solution set. Let y_1 be covered by [x, x + 1] for some x with $x < y_1$. Since y_1 is the smallest, [x, x + 1] only covers points after y_1, and so we can replace [x, x + 1] with $[y_1, y_1 + 1]$. This shows that the greedy choice works.
5. What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers: a:1, b:1, c:2, d:3, e:5, f:8, g:13, h:21? Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers.
    - a = 0000000 (7 0's)
    - b = 0000001
    - c = 000001
    - d = 00001
    - e = 0001
    - f = 001
    - g = 01
    - h = 1
    - Generalization. Let H[i] denote the Huffman code of the i-th Fibonacci number. Then H[n] = 1, H[n-1] = 01, H[n-2] = 001, ..., H[2] = $0^{n-3}1$, H[2] = $0^{n-2}1$, H[1] = 0^{n-1}.