Name:

Roll no.:

There are ~~15~~ 16 questions in total, worth 100 ~~105~~ points. For **Q1 to Q14** only brief 3-4 line answers are required. Write answers in BLACK right after every question in the space provided. *lg means* log_2.

Q8 had an error -- that question is not included.

Q1. [2+5 points] What is the tightest upper bound on T(n) if it satisfies the recurrence relation

T(n) = T(n/2) + T(n/3) + T(n/6) + n

(a) O(1), i.e., <= c for some constant c
(b) O(lg(n)), i.e., <= c lg n for some constant c
(c) O($\sqrt{n}$), i.e., <= c $\sqrt{n}$
(d) O($n^{2/3}$), i.e., <= $cn^{2/3}$
(e) O(n), i.e., <= cn
(f) O(n lg n) …
(g) O($n^{3/2}$) …
(h) O($n^{3/2}$ lg n) …
(i) O(n^2) …
(j) O(n^3) …
(k) None of the above: Specify the bound _____

Prove your answer using induction. For full credit, also provide a value of c. *(You may need a calculator for this question, and it is allowed, only for this question.)*

Proof:

Assume that T(n) <= n lg n for n <= m-1.
We have to prove that T(m) <= m log m.

T(m) <= T(m/2) + T(m/3) + T(m/6) + m
<= (m/2)(lg m - 1) + (m/3)(lg m - lg 3) + (m/6)(lg m - lg 6) + m
= m lg m + m(1 - ½ - (lg 3)/3 - (lg 6)/6)
= m lg m + m(1 - (lg 2)/2 - (lg 3)/3 - (lg 2)/6 - (lg 3)/6)
= m lg m + m(1 - (⅔)lg 2 - (½)lg 3) = m lg m + m (1 - ⅔ - 1.585/2)
= m lg m + m(⅓-0.79…) < m lg m

Q2. [5 points] The 4SUM problem takes as input a set of real numbers and asks if there are four numbers whose sum is 0. The 3SUM problem similarly asks if there are three numbers whose sum is 0. Consider this algorithm to reduce from 3SUM to 4SUM.

```
def reduce3SUMto4SUM (set S):
      return S U {0} // add 0 to the set
```

State the correctness lemma that this algorithm should satisfy for it to be a correct reduction. Then either prove or disprove this lemma.

[2.5 points] Lemma: {a1 a2 … an} has 3 numbers whose sum is 0 iff {a1 a2 … an 0} has 4 numbers whose sum is 0.
[2.5 points] The lemma is incorrect. Suppose S = {-1,-2,1,2}. Sum of all elements of S is 0. Hence, the sum of any three is never 0. So S does not have 3 numbers that add to 0. However, {-1,-2,1,2,0} has 4 numbers {-1,-2,1,2} that add to zero. This disproves the lemma.

Q3 [5 points] Does this algorithm compute a constant-factor approximation to the largest independent set of an input graph? You may require the fact that if V' is a vertex cover of G=<V,E> then V-V' (vertices *not* in V') form an independent set, and vice versa.

```
def approxIS(G=<V,E>):
    T = minimum spanning tree of G
    V' = internal nodes of T
    return V-V' // return the complement of V'
```

Ans:

No. Consider this infinite family of graphs: for any n, Gn is a line/path-graph with n nodes and edges only between v1-v2-v3... -vn. For such graphs, T is the same as G, and so approxIS will return 1 node. However, the largest independent set consists of alternating nodes whose number is at least n/2. Hence, there is no constant c such that |approxIS| >= |OptIS|/c.

** **Q4 [5 points]** The SMALLFACTOR problem takes as input a number n in binary and returns true if n has some factor f that is less than ⌊lg(n)⌋, else returns false.

Is SMALLFACTOR a P problem, an NP problem, an NP-hard problem, an NP complete problem? Explain briefly.

Ans:

[40% for algorithm, 60% for complexity]

This is a P problem, and hence, also in NP. An algorithm can try all numbers from 2 to floor(log(n)) (there are log(n) many such numbers) and for any such number, say x, find out if x divides n. Instead of trying a polynomial time division algorithm (which we did not study), we can try to binary search a y from 2 to n (we have to make log(n) searches) such that x*y=n; for every such y, we can use the integer multiplication algorithms that we studied to verify if x*y=n. Since both x and y are log(n) bits, the multiplications will take poly(log(n)). So the total complexity is: log(n)*log(n)*poly(log(n)) which is a polynomial in the number of bits in n.

Q5 [5 points] Consider this algorithm that tries to determine the chromatic number of a planar graph using the fact that every planar graph can be coloured using at most 4 colours (*The story behind the proof of this fact is worth reading later.).*

```
def PlanarChromatic(planar graph G):
      if G has no edge: return 1
      else if G is bipartite: return 2
      else return 4
```

(a) What is the relative approximation ratio of this algorithm? Explain.

Ans: **[3 points]** 4/3 is the relative approximation ratio.
If the graph has no edge or is bipartite, then OPT=APPROX.
For all other planar graphs, OPT is 3 or 4, and APPROX=4.
So for all cases, OPT <= APPROX <= OPT*(4/3).

(b) What is the absolute approximation ratio of this algorithm? Explain.

Ans: [2 points] 1 is the absolute approximation ratio.
If the graph has no edge or is bipartite, then OPT=APPROX.
For all other planar graphs, OPT is 3 or 4, and APPROX=4.
So for all cases OPT <= APPROX <= OPT+1.

Q6 [5 points] Let $[a_0,a_1, \dots a_{n-1}]$ denote the values of a degree (n-1) polynomial Y() at the n n-*th* roots of unity. Fill in the blanks to complete the algorithm that computes an inverse discrete Fourier transform (IDFT) of order n via a O(n log n) algorithm.

def $IDFT_n([a_0,a_1, \dots a_{n-1}])$:
  if n=1: return __[a0]__ **[1 point]**_____
  else:
    $[y_0^{even},y_1^{even}, \dots y_{n/2-1}^{even}] = IDFT_n([a_0,a_2, \dots a_{n-2}])$
    $[y_0^{odd},y_1^{odd}, \dots y_{n/2-1}^{odd}] = IDFT_n([a_1,a_3, \dots a_{n-1}])$
    for k=0 … n-1:
      y_k = ___$(1/n)[y_{k \bmod n/2}^{even} + e^{2\pi ik/n} y_{k \bmod n/2}^{odd}]$**[4 point]**
    return $[y_0,y_1, \dots y_{n-1}]$ // coefficients of Y()

Write an algebraic expression in each of the blanks above (not a pseudocode/algorithm/multiple statements, etc.). No explanation is needed, but you should know what you are writing.

Q7 [3+5=8 points] Suppose we want to obtain a 1000/999-relative approximation algorithm to the solution of a Knapsack instance and we run the scaling-based approximation algorithm that was studied in some lecture. Answer the following questions in terms of these values: n is the number of items, v_{min} is the smallest value, v_{max} is the largest value, W is the Knapsack weight limit, w_i is the weight of the i*th* item.

(a) In the scaling-based algorithm, the original value of each item, say v_i of item i, is modified to some v'_i. Write the formula to compute v'_i from v_i.

Ans: $v'_i = floor(1000*n*v_i/v_{max})$

(b) After scaling, the Knapsack instance with the modified values is solved using a dynamic programming algorithm. What would be the running time of this algorithm in terms of n and W (it is not necessary to use botn n and W, but you cannot use anything else). Explain your answer.

Ans: $T = O(n * [v'_1 + v'_2 + \ldots + v'_n])$

Also, $v'_1 + v'_2 + \ldots + v'_n$

$<= 1000*n*v_1/v_{max} + 1000*n*v_2/v_{max} + \ldots + 1000*n*v_n/v_{max}$

$<= n*1000*n*1$ (since $v_i <= v_{max}$)

$<= 1000*n^2$

So $T = O(n^3)$

*** **Q8. [5 points]** We studied bin packing in a tutorial in which the input is a set of n items, with weights denoted by W={w1 … wn} The problem is to compute the minimum number of bins with weight capacity C that can fit all the bins. Suppose that there is a polynomial time algorithm BPAlgo that returns a 1.25 ~~1.3~~-relative approximation to the optimal number of bins. Design an algorithm that uses A and solves the PART3TION problem in polynomial time. The PART3TION problem takes as input n integers {a1 … an} and asks if there is a way to partition A into 3 disjoint subsets A1 A2 and A3 with equal sums. Briefly explain the correctness and complexity of your algorithm. Errata: I had written 5/4 originally, which I replaced by 1.3 during typing. My mistake.

```
def SolvePart3Tion(A={a1 … an}): // array of integers
      S = a1 + a2 + … + an
      If S is not divisible by 3 or if S/3 is not an integer:
            return false
      if BPAlgo(W=A, C=S/3) = 3:
            return true
      else:
            return false
```

Time complexity is O(n) + Time(BPAlgo)) = poly(n).

A can be partitioned into 3 subsets with sum S/3 iff the number of T-capacity bins to fit A is 3.

If the number of optimal bins = 3, then BPAlgo returns 3 (1.25*3 = 3.75 is not an integer and 3 is the next lowest). If the number of optimal bins = 4 or more, then BPAlgo returns 4 or more.

Q9. [5 points] We call a graph ``compact' if there is a path with at most 2 edges between any two vertices in the same connected component.

```
def reduce(G): // G is any unweighted undirected graph
    G' = copy of G
    Add a vertex v to G'
    Add edges from v to every vertex
    Add another vertex u and add edge (u,v)
    Return G'
```

```
Lemma: G has a Hamiltonian path iff reduce(G) is compact and has a
Hamiltonian path.
```

Fill in the blanks above such that reduce() is a polynomial-time algorithm and it satisfies the lemma below. Explain how both these criteria are satisfied.

Ans:

Reduce() adds 2 vertices and n+1 edges to a n-node graph. Hence, it takes polynomial time.
G' is compact since (1) u can reach v with 1 edge and reach every other vertex w within 2 edges, u to v and v to w, (2) v can reach every vertex with 1 edge, (3) all other vertices can reach any other vertex within 2 edges by going through v.
If G has a Hamilton path p then G' has this Hamiltonian path: u - v - p
If G' has a Hamiltonian path p then p must start at u (since u has degree 1), go to v, then visit all the other vertices. The subpath after v is then a Hamiltonian path of G.

Q10. [5 points] Complete the following divide and conquer algorithm to find all elements in an n-sized integer array that appear n/5 or more number of times. Note that there can be at most 5 such elements. Explain your approach and discuss the time complexity of your algorithm.

```
def FindFrequent(A[1...n]): // returns at most 5 numbers
    AL = A[1… n/2]
    AR = A[n/2+1 … n]
    (u1,u2,u3,u4,u5) = FindFrequent(AL)  // some of these could be Null
    (v1,v2,v3,v4,v5) = FindFrequent(AR)   // some of these could be Null
    i=1, w1=w2=w3=w4=s5=NULL
    for each element x in {u1,u2,u3,u4,u5,v1,v2,v3,v4,v5}:
        t = number of copies of x in A
        If t >= n/5: wi = x, i=i+1 // x is frequent element
    return (w1,w2,w3,w4,w5) // some of these could be Null
```

Explanation and analysis:

Recurrence T(n) = 2T(n/2) + O(n) // the loop makes a linear pass and the loop is over at most 10 elements. Solution of recurrence : O(n log n)
If x is a frequent element, then frequency of x in A >= n/5. However, if freq(x in AL) < (n/2)/5 and freq(x in AR) < (n/2)/5 then freq(x in AL+AR) < n/5. So, x should appear at least 20% of the time in at least one of AL or AR. The algorithm identifies all elements that appear at least 20% of the time in both AL and AR and checks which of them appear at least 20% of the time in A by calculating their frequency making linear passes over the array.

Q11 [5 points] In HW23, you were asked to implement a data structure using the disjoint-set API to maintain an array of bits X[1 ... n] along with these operations.

(i) Init() : Initialization. All bits of X are initialized to 0.
(ii) Set(i) : Set X[i]=1. It is guaranteed that i <= n-1 so the rightmost bit of X[] is never set.
(iii) IsSet(i): return X[i]
(iv) NextUnset(i): return the next largest smallest j >= i s.t. X[j]=0. This will always return some index.

Which disjoint-set implementation will you choose, and why, if you require that Set() has ammortized complexity O(log n) and NextUnset() has worst-case complexity O(1)? Explain.

Ans: Set() is implemented using 1 Union() and NextUnset() is implemented using 1 Find().
We should use the shallow-tree+threading+union_by_rank implementation. The ammortized cost of Union() in this implementation is O(log n) and the cost of Find() is O(1).

Path-compression+union-by-rank is not suitable here since its Find() has O(log n) worst-case complexity.

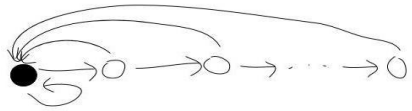
Q12. [5 points] Suppose I am trying to find the shortest distances from a source vertex s on a graph G using the Bellman-Ford algorithm. While filling the memo OPT[i,v], where i is an integer and v is a vertex, in the increasing order of i (starting at i=0 and filling the table row-wise), I observe that for one particular k, OPT[k-1,u] = OPT[k,u] for all vertices u.
Suppose dist(s,u) denotes the shortest distance between s and u. Which of the following is true and why?

(a) OPT[k,u] < dist(s,u) (b) OPT[k,u] <= dist(s,u) (c) OPT[k,u] = dist(s,u)
(d) OPT[k,u] >= dist(s,u) (e) OPT[k,u] > dist(s,u) (f) None (answer depends on G)

Ans: The basic idea is the recurrence formula for OPT[j,.] depends only on OPT[j-1,.]:
OPT[j,u] = max_v { OPT[j-1,u], OPT[j-1,v] + w(v,u) }

So, the values for any j (i.e., the j-th row in the memo storing the OPT values) can be filled entirely from the values for j-1 (the (j-1)th row). Since the k-1 and k-th rows are identical, this would mean the (k+1)-th row would be the same as them, and so will be all the next rows, including the (n-1)-th row. The (n-1)th row stores the distances dist(s,u). So dist(s,u)=OPT[n-1,u]=OPT[k,u].

Q13 [5 points] Consider an implementation of disjoint-sets using lists & pointers as illustrated below. Leader of a node is identified by the presence of a self-loop on a node.



This implementation is the same as the shallow-tree+threading. So the complexities follow from what was done in class.

(a) **[2]** What is the worst case complexity of Union()? Explain briefly. **Ans**: O(n) since if both the sets have O(n) elements each, then it will take O(n) time to update all the leader pointers of one of the sets.

(b) **[1]** What is the worst case complexity of Find()? Explain briefly. **Ans**: O(1) since the algorithm has to simply follow the leader/parent pointer once to find one's leader.

(c) **[2]** What is the ammortized complexity of Union()? Explain briefly. **Ans**: O(log(n)) using union-by-size. The leader of the larger set becomes the leader of the united set. It was shown in class that when the leader of an element changes the size of the set containing that element is at least doubled; hence, the leader of any element can only change log(n) times. Therefore, for all the n elements, the total number of times the leader pointer changes is O(n log n) giving an average complexity of O(log n).

*** **Q14 [5 points]** Another homework scheduling problem. Suppose that you have been assigned n questions for the endsem exam where each question is announced and due at different times - question Qi is announced at time start(i) and is due at time due(i). mark(i) denotes the marks that you know you will get for Qi (you have figured out how to satisfy the TAs by now). But you also know that you can only work on one problem at any time, and once you start working on a question you will be busy solving it until its due time comes, and at that point you will submit it. Call this problem SCHEDULE which takes as input three arrays, start[1...n], due[1...n], mark[1...n] corresponding to the n questions. which outputs the maximum marks you can score by cleverly choosing which questions to solve. For simplicity, assume that the questions are sorted in increasing order of their due times.

Is SCHEDULE NP-hard? Explain your answer.
If you say Yes, briefly explain a reduction from some NP-hard problem (state the problem clearly). You do not need to state or prove the correctness lemma.
If you say No, briefly explain a polynomial-time algorithm using dynamic programming (write only the first two steps -- definition of a subproblem and recurrence for it). <- hidden hint.
Ans:
This can be solved in polynomial time using dynamic programming.

Q15 [15 points] You are given a sequence S[1...n] of integers and a number k. The objective is to partition the sequence into k non-empty subsequences. The "value" of a subsequence is the sum of the integers in that subsequence and "bottleneck" of a partitioning is defined as the minimum value among all the corresponding subsequences. The partitioning should be done in a manner that *maximizes* the bottleneck.

The following partitioning has bottleneck = min{2+3+1+4+5, 3+1+..., 2+6+4+..., 34+5+19} =15.
2,3,1,4,5, 3,1,3,4,6,3,6,4,3, 2,6,4,3,2,5,4,6,3, 34,5,19

Design a polynomial-time algorithm for this problem using dynamic programming. Explain all the 6 steps. *Hint: I know one that uses a 2-dimensional array.*

Ans: I use Val(a,b] = S[a] + S[a+1] + … + S[b-1] + S[b] (the value of a subsequence).

1) **[3 points]** OPT[i,j] = minimum value possible from S[1...i] by breaking it into j non-empty subsequences.

2) **[4 point]** if i < j OPT[i,j] = infinity // less than j symbols cannot be partitioned into j non-empty portions.
o/w OPT[i,j] = max_{b=j...i} OPT[b-1,j-1] + Val(b,j) // last subsequence is from b...j and 1…(b-1) should be partitioned optimally into j-1 non-empty subsequences.

[2 point] Base case: OPT[i,1] = Val(1,i) // entire S[1...i] forms the subsequence

3) **[1 point]** OPT[n,k] gives the optimal bottleneck in partitioning S[1...n] into k non-empty subsequences.

4) **[1 point]** OPT[i,j] values can be stored in an (n x k) 2D array.

5) **[1 point]** To compute OPT[.,j], OPT[.,j-1] values are needed. Hence, the memo can be filled columnwise, starting with OPT[.,1] and moving right. Each column can be filled in any manner. The final answer is at bottom right.

6) **[1 points]** Time complexity is $O(n^2k)$ since the memo has nk entries and filling any entry requires taking the maximum of at most n entries (b can take at most n values in the recursive formula).

[2 points] Space complexity is O(n) since we can store only two columns.

Q16. [15 points] Consider a variation of the above optimization problem (stated in **Q15**). In the variation you have to partition the sequence into smaller, not necessarily contiguous, sequences. Define a decision problem SEQPART corresponding to this optimization problem and prove that SEQPART is an NP-complete problem.

Ans:
[3 points] SEQPART is defined as: On input a set S of integers, an integer k and a threshold b, is there a partitioning of S into k non-empty sequences with a bottleneck >= b?

Proof of SEQPART being an NP problem:
[2 points] def verify(S,k,b,P): // P is an array of partitions of S
    if P is not a correct partitioning of S into k non-empty sequences:
        return false
    B = minimum of bottleneck values of the partitions in P
    if B >= b: return true
    else: return false

[1 point for showing verify() is polytime] The algorithm has to verify if P is a correct partitioning, which can be done using k passes of S and has to compute B which can be done by a linear pass over P. Both steps take O(n).

Proof SEQPART being NP-hard:**[1 point for stating reduction from what]** Reduction from PARTITION.

[4 points for reduction] def reduce(array of integers A): // A is an instance of PARTITION
    Total = sum of values in A
    Return (A,2,Total/2)

Reduction algorithm has to compute the sum of the elements in A which can be done in O(n). Hence the reduction is polytime.

Lemma: An array A can be partitioned into two subarrays with equal sums iff A can be partitioned into two subarrays with bottleneck >= Total/2.

Note that “A can be partitioned into two subarrays with equal sums” is equivalent to “A can be partitioned into two subarrays with sum Total/2”.

[2 points] Proof of => direction: If A can be partitioned into two subarrays with sum Total/2, then the bottleneck of that partitioning is Total/2.

[2 points] Proof of <= direction: If A can be partitioned into two subarrays with bottleneck >= Total/2, then each subarray must have value >= Total/2 (since bottleneck is the minimum value among all the subarrays). This can only happen if each subarray has value = Total/2 and this implies that A can be partitioned into two subarrays with equal sums.