Introduction to Algorithms	Review information for Final
CS 4820, Fall 2019	with Solutions

Rooms, Time, and Review for Final

- Final is Thursday, December 19th 7-9:30pm.
- We have two rooms:
  - If your last name starts between A-L then your exam will be in Uris Hall G01 (our usual lecture hall),
  - If your last name starts between M-Z then you need to go to Statler Hall 185-Auditorium (downstairs only).

  People taking the exam in not the assigned room, lose at least 10% of the credit.
- **If you need special arrangement, have a conflicting exam, or have 3 exams in a 24 hour period, you should have contacted Corey Torres (ct635).**
- We'll have multiple review sessions by TAs on the 12th and the 16th. Exact times will be announced on Piazza.
- Solutions to the practice questions will be released on Monday, Dec 16 via CMS, but you can use office hours to discuss solution with the TAs any time before the prelim.

Guidelines for the Final

The final is cumulative, covering all topics from the beginning of the course. It will be closed book and closed notes.

In particular, the topics include

- From Prelim 1: (To review material, see the review packets from the prelim and homeworks 1-4. New practice questions are included below.)
  - Stable matching from Chapter 1
  - Greedy algorithms from Chapter 4, including unweighted interval scheduling, minimum spanning tree (Kruskal, Prim, and Boruvka), and Huffman coding.
  - Dynamic programming from Chapter 6, including Weighted interval scheduling, segmented least squares, sequence alignment (a.k.a. edit distance), Bellman-Ford shortest path algorithm.
  - Divide and conquer from Chapter 5, including integer multiplication, convolution (sec 5.5, but only to recognize a convolution problem, not the details of how to run the algorithm)
  - Randomized Algorithms from Section 13, including expected linear-time median finding, hashing, and prime testing (see notes on the web, plus the homework question on problem set 5)

- From Prelim 2: (To review material, see the review packets from the prelim and homeworks 5-7. New practice questions are included below.)
  - Network flows from Chapter 7, including the Ford-Fulkerson max flow algorithm, equivalence of max flows and min cuts, and network flow applications.
  - NP-completeness from Chapter 8, including reductions and the problems SAT, Independent Set, Vertex Cover, Set Cover, Hamiltonian Path/Cycle, TSP, and problems from the homeworks/exams
- Since Prelim 2: (see summary and review questions below, as well as homeworks 8-9.)
  - Computability: see notes on the web site, including the non-computability of halting problem, and reductions.
  - Approximation Algorithms from Chapter 11 and the notes posted, including vertex cover via linear programming, the greedy knapsack algorithm from 11/22 (see notes on the web site), set cover.
- Topics we covered briefly, but are not included are Turing Machines, details of prime testing algorithm, details of the convolution algorithm, and the last two lectures: Center Selection and the Algorithms and Selfish Users.

Computability

The questions on computability will test your ability to decide which problems are computable, and which are not. Topics you need to know about:

- definitions of decidability and recognizability from problem set 8,
- the halting problem, co-halting problem, and accept problem,
- judging whether a problem is decidable, or regonizable, and
- proving a problem is undecidable or unrecognizable by reduction from an undecidable or unrecognizable problem.

Approximation Algorithms

- greedy methods: such as knapsack (see handout), or Set-cover approximation algorithms (Sec 11.3)
- linear programming and rounding, (Section 11.6 and problem 3 on ps9)

Practice Questions for the Final with Solutions

A good way to review for the final is to try solving all the homework problems again *without looking at the solutions*, including your own solutions. This serves as a great reminder for what things we expect you can do at the end of the course. We also suggest that you review the comments the TAs left for your solutions, and the alternate solutions and common mistakes we posted with the solutions on CMS.

The following sample questions are designed to give you a sense of the type of questions you should expect on the prelim. The questions are from previous years, you may want to try them under prelim

conditions. The first four were last years final. Solutions to the question will be posted on CMS on Dec 16th. You can also discuss the questions with the TAs in office hours.

If you would like to try more practice problems, the first few exercises at the end of each chapter are an excellent resource for this. You can use Piazza to ask us about solutions to these problems.

1. *(32 points)* **Short answer.** Each of these questions asks for a true/false or decidable/undecidable answer and a short explanation. Follow the instructions for each question. **Detailed proofs are not required in this section.**

a. *(8 points)* Consider an undirected graph $G = (V, E)$ with integer costs $c_e \geq 1$ for each edge $e \in E$, and a minimum-cost spanning tree T in this graph. Suppose we subtract 1 from the cost of each edge: $c'_e = c_e - 1$ on all edges. *True or False*: does T remain the minimum cost spanning tree. If true, give a brief explanation; if false, give a counter example.

Solution: True. The choice of the MST only depends on the sorted order of the edge weights by Kruskal's algorithm, and decreasing all costs by 1 doesn't change the order.

b. *(8 points)* Consider an undirected graph $G = (V, E)$ with integer costs $c_e \geq 1$ for each edge $e \in E$, and a shortest path P between two nodes $s, t \in V$ in this graph. Suppose we subtract 1 from the cost of each edge: $c'_e = c_e - 1$ on all edges. *True or False*: does P remain the shortest path from s to t? If true, give a brief explanation; if false, give a counter example.

Solution: False. Consider a four node graph with nodes s, v, w, t and costs $c_{sv} = 2, c_{vw} = 2, c_{wt} = 2$ and $c_{st} = 5$. Now the shortest path is the one edge direct path $P = (s, t)$. Subtracting one from the cost of each edge changes the shortest path to $P' = (s, v, w, t)$ which now has cost 3.

c. *(16 points)* You are editing some old code at an internship, and notice a line of code that looks odd. You wonder if this instruction will ever be executed, or if it can be deleted. More formally, the Executes Line problem is: given the Java program M and a particular instruction on line k of this program, is there any input x on which the program M will execute the instruction in line k?

Is this problem decidable? Explain your answer. If the problem is decidable or give the algorithm to decide it (but not need to prove that your algorithm is correct). If it's undecidable give the reduction (but not the proof of correctness) .

Solution using reduction: Not decidable. We reduce to the Halting Problem to our problem. Suppose we have a decider. Consider an input M, x to the `HaltingProblem`. Write a wrapped program M' using M and x as follows. M' ignores its own input, and writes input x into memory and then simulates M on x. Then add a new line (e.g., setting a new variable to 0). Now we have that M' ever executes this new line we just added, if and only if M halts on x.

2. *(20 points)* **Algorithm Design.** You are a contestant on a new obstacle course game show, "America's Got Obstacles." Your goal is to traverse an obstacle course in the coolest way possible, where judges will add or remove points from your score based on what technique you use to get around each obstacle.

Assume that there are n obstacles in the obstacle course. At each obstacle i, you have a choice: either you can perform a technique that gets you past obstacle, for which you believe you will get a_i points (where a_i can be positive or negative), or you can perform a more complicated stunt that will allow you to traverse both obstacles i and $i+1$, giving you b_i points (again, this may be positive or negative). You cannot then go back to obstacle $i+1$; your next technique will start from obstacle $i+2$, and need to continue till you get past all obstacles.

Give an efficient algorithm that returns the most points you can collect reaching the end of the obstacle course. *You do not need to prove the algorithm's correctness or prove a reduction correct, but you should analyze its running time, which should be polynomial in the number of obstacles n.*

Solution by new DP: Table: Describe $OPT(i)$ as the maximum score to get over obstacle i.

Base case: $OPT(0) = 0$, where you start. $OPT(1) = a_1$, as it can only be reached by a single-obstacle move.

Recursive case: $OPT(i) = \max(OPT(i-1) + a_i, OPT(i-2) + b_{i-1})$. We argue that to end at obstacle i requires either progressing from the previous obstacle with score a_i for the technique, or from the obstacle before that with score b_{i-1}. for the technique. For each of these cases, the optimal total route is the combination of the optimal route to one of these two points combined with the corresponding option of a or b.

Solution: The total score is in $OPT(n)$.

Running time: At each step, our computation only considers up to two options, which takes constant time. Our table has dimension n, so our running time is $O(n)$.

Note: the backwards way also works!

Table: Describe $OPT(i)$ as the maximum score to get from obstacle $n-i$ to n. (Or some other indexing)

Base case: $OPT(0) = 0$, $OPT(1) = a_{n-1}$

Recursive case: $OPT(i) = \max(OPT(i-1) + a_i, OPT(i-2) + b_i)$

Solution: The total score is $OPT(n)$.

Running time: same as above.

Solution by reduction to SLS: Compute an error table where $e_{i,i+1} = -a_i$, $e_{i,i+2} = -b_i$, and everything else is $e_{i,j>i+2} = \infty$. Each move on an obstacle is a line segment. The solution value will be the negative solution to the segmented least squares problem with this precomputed error table. The table takes $O(n^2)$ to construct, as does solving SLS, so the total running time is $O(n^2)$.

Solution by reduction to weighted interval scheduling First, observe that we have to make intervals non-negative, which we do by finding the minimum value of a_i and b_i (call it m) and subtracting it. Make sets of intervals of length 1 and value $a_i - m$ starting at time i, and intervals of length 2 and value $b_i - 2m$ starting at time i. After solving this interval problem, the solution requires adding back to the solution to the weighted interval problem $OPT(n) + mn$. This takes $O(n)$ time to find m, $O(n)$ time to convert the problem into one with the intervals in sorted order, $O(n)$ time to get the right predecessor values precomputed, and $O(1)$ time to subtract out mn, so it runs in $O(n)$.

Solution by reduction to shortest path Because this has no cycles, we don't have to worry about negative distances, so similar to SLS, $d_{i,i+1} = -a_i$, $d_{i,i+2} = -b_i$, and all other distances are ∞. The solution value is the negative value of the shortest path solution. The distance table takes $O(n^2)$ time to construct and the shortest path problem takes $O(n^2)$ time to run, so the combined running time is $O(n^2)$.

3. *(18 points)* **Approximation Algorithm.** Consider the **Move-It-Quik** truck problem from problem set 10 (from last year, see details here): Recall that items arrive one at-a-time and the truck weight capacity is K pounds. The loading site only has room for one truck to be loaded at a time. Rather than planning which items should go in which truck in advance, the movers just load one item at a time in whatever order they find items to pack, disregarding their weight, until the next item wouldn't fit. Then, they move on to the next truck.

The pseudocode for the truck-loading algorithm from problem set 10 is as follows:

```
TRUCK LOADING:
    c = K is the capacity remaining for the current truck
    m = 1 is the number of trucks used so far.
    While there is an item i with weight w_i left to load:
        If c ≥ w_i:
            // The item fits; load item i on the current truck
            Set c := c − w_i
        Else:
            // The item won't fit; send off the current truck
            // and get a new one to load item i
            Set m+ = 1
            Set c = K − w_i
        Endif
    Endwhile
    Return m as the total number of trucks used.
```

Suppose at the end of one of these moving jobs, n items with weights $w_1, w_2, \ldots, w_n$ are packed up in that order. In the homework, you were asked to proved that the number of trucks used by this algorithm, m, is at most a factor of 2 larger than the minimum number possible for taking the n items. That is, if OPT denotes that minimum number of trucks possible by carefully ordering the n items, we showed that $m \leq 2OPT$.

For this problem assume that the system has no items that are too heavy. Concretely, assume that $w_i \leq K/3$ for all i. Prove that in this case the online packing of the above algorithm satisfies a better bound of $m \leq 1.5OPT + 1$

Solution: By the assumption that $w_i \leq K/3$, we won't start a new truck unless the previous one has at least $2K/3$ weight. So with m trucks the total weight must be more than $(m-1)2K/3$, and hence we need more than $OPT > \frac{2}{3}(m-1)$ trucks, implying that $m < \frac{3}{2}OPT + 1$.

4. *(30 points)* **NP-completeness.**

You are planning a deep-dish pizza party for an event. You are currently trying to agree on a set of pizzas that will be available for the g guests to order. Each type of pizza offered on your menu can have up to 2 toppings on it out of a set T containing t possible toppings.

Each guests i has the following constraints:

- they have a non-empty set $L_i \subseteq T$ of toppings that they like, and will not eat a slice of pizza *without at least one* of these toppings, and
- they have a set $H_i \subset T$ of toppings that they hate, and will not eat a slice of pizza *with* any of these toppings.
  It is possible for a topping j to be in neither L_i nor H_i, but it cannot be in both L_i and H_i.

Given the number of guests g, the toppings T, and the preferences of toppings L_i and H_i for each guest, is it possible to come up with a menu containing only p or fewer unique types of pizzas (each with no more than 2 toppings) such that every guest will be okay ordering at least one of the pizzas? You should assume that you will have unlimited supply of each type of pizza. We call this the PizzaOrdering problem.

Prove that the PizzaOrdering *problem is NP-complete.*

Hint: Consider the version where pizzas can only have one topping. Proving hardness with this version only will be a maximum of 20 points on the problem.

Solution: This problem is NP-complete. First, to show it is in NP, we can give a certificate of the list of p pizzas and the 2 toppings corresponding to them, followed by the list of which pizza each guest could order. This takes space $O(p + g)$. Verifying requires checking that each pizza is valid in its ingredient list and that each guest is compatible with their pizza, which performed extremely naively takes time $O(tg)$ to check through the preferences for each guest on a single pizza's toppings.

To show that the 1-topping version is NP-complete, we can ignore the H_i lists. We can do a reduction from SetCover. Input to SetCover is a list of sets $S_i \subset U$ for $i = 1, \ldots, n$, and an integer k, and the question is if there is a set $I \subset \{1, \ldots, \}$ such that $|I| = k$ and $\cup_{i \in I} S_i = U$. We define a guest for every element $u \in U$, and a topping for each set $i = 1, \ldots, n$. Guest u likes pizza topping i if and only if $u \in S_i$, that is S_i is the the set of guests who like topping i. No guest hates any topping. We claim that k pizzas with one topping each will keep all guests happy if and only if there are k sets whose union is U.

To prove the problem with 2-toppings is NP-complete, we can reduce from Set Cover or from Vertex Cover. Using SetCover, modify the above reduction to make each guest g hate all toppings i such that $u \notin S_i$. We write out a more detailed proof for Vertex Cover, though the same proof works for the above construction for Set Cover also.

The input for Vertex Cover is a graph $G = (V, E)$ and a number of vertices allowed in the cover k. Convert each vertex $v \in V$ into a topping in T (so $t = |V|$), and each edge $e \in E$ into a guest (so $g = |E|$). If edge $e = (a, b)$ is turned into guest i, L_i will only contain the two toppings corresponding to the vertices a and b at the endpoints of that edge, and H_i will contain all other toppings. Set $p = k$. This is a polynomial-time reduction: $V, E, and k$ can be copied directly into $T, g, and p$, and to make each set of L_i and H_i takes $O(|V|)$ time (so $O(|V||E|)$ total).

To prove that if there is a vertex cover, there is a pizza order: Convert each of the k vertices in the cover into a 1-topping pizza containing the topping corresponding to that vertex. Every edge must have one endpoint covered by one of the k vertices; corresponding to this, every guest i must have some topping from the set of vertices in their L_i.

To prove that if there is a pizza order, then there is a vertex cover: Consider a collection of pizzas solving this. If any pizza in this order has more than 1 topping, we can reduce it to an order with only

one topping per pizza by observing that a 2-topping pizza can only satisfy one guest (as only the guest whose endpoints are the two pizza toppings won't hate one of the toppings), in which case that pizza would still satisfy that guest if it only had one of those two toppings. If we discard a random topping from any 2-topping pizza, we will therefore have a pizza menu where every pizza only has one topping. Choose the vertices corresponding to those single toppings; because guests only like toppings if and only if their corresponding edges in the original graph had endpoints at vertices corresponding to those toppings, these toppings must correspond to a VC in the original graph.

Hardness with 1 topping only: This is a lot easier, as its really a case of hitting set: we need to select k toppings S such that $S \cap S_i$ non-empty for all i. This is a special case of the problem on the prelim called ProjectorHelper.

We can also simplify this from VertexCover or SetCover: the reduction is the same as before, where edges/elements become guests, vertices/sets become toppings, and the guests list L_i is based on what edges they touch/what sets they are in from the corresponding VertexCover/SetCover problem, with k the maximum number of vertices/sets as the number of unique pizzas on the menu. In this case, however, the list H_i doesn't need to be set at all. The proof is a simpler version of the one above: if all guests are satisfied, it's because there was a set of toppings corresponding to the definition of a Vertex Cover or Set Cover. If there is a vertex cover or set cover, using one-topping pizzas corresponding to the cover will satisfy all guests.

They may choose to reduce from their 1-topping reduction to 2 toppings. Here, you just do the last trick: add everything not in L_i to H_i. This also takes polynomial time. A 1-topping solution from before will still exist in the 2-topping setting, as every pizza had to satisfy only people who liked (and thus didn't hate) the topping. Any 2-topping solution must also be convertible into a 1-topping solution: if the guests satisfied didn't agree on both toppings on the pizza, then one would hate one of the toppings, so one topping must be redundant.

Reductions from SetCover or VertexCover as the base problem as same as they were at the prelim: Sets correspond to toppings, elements to guests, and the set L_u that a guest u likes are sets $\{i : u \in S_i\}$.

5. For the problem below provide one of the following two answers:

- There is a polynomial-time algorithm for the problem; or
- the problem is NP-complete.

Explain why your answer is correct.

You need to select a set of k judges for a competition. Unfortunately, all judges know some of the participants socially, which represents a form of conflict of interest. Each possible judge i had to declare the list L_i of all the participants with whom he or she has a mild conflict of interest. There are so many conflicts that it turned out to be impossible to avoid them all. One option for how to deal with them is as follows.

To minimize the effect of these conflicts, you would like to select k judges so that for any participant j at most one of the selected judges has a conflict of interest with j. We call this the Judge Selection Problem.

Is this form of the Judge Selection Problem polynomial time solvable or NP-complete?

Solution. NP-complete. It is clearly in NP; checking if a set of k judges have this property is easy.

To prove it is NP-complete, we show that INDEPENDENT SET $\leq$ JUDGE SELECTION PROBLEM. Input to INDEPENDENT SET is a graph $G = (V, E)$ with k. We let the nodes V be the judges, and the edges be the participants. each judge has a conflict of interest with the edges adjacent to the node. There are k judges with the property claimed, if and only if G has an independent set of size k.

- If G has an independent set of size k, the judges corresponding to the nodes satisfy the required property, as no two of them are adjacent to a shared edge.
- If there is a set of judges with the property above, then the set of nodes corresponding to the judges forms an independent set.

Construction is done in linear time.

6. Determine whether the following decision problem is decidable, recognizable but not decidable, or not recognizable. Explain why your answer is correct.

Given a program M and two input strings x and y. Does M output the same result for x and y? That is, is it the case that, if M accepts, rejects or doesn't stop on x, does it do the same for y?

Solution. It is undecidable. To show this, we show that if we could decide this problem, the decision algorithm (we will call it EquivalentInput(M, x, y)) could also decide the halting problem, which is a contradiction.

Consider an input to the halting problem (M, x), aiming to decide if M accepts x. Now we turn this to input to the above decision problem.

Let $M'(y)$ be the program that first executes $M(x)$, then checks if y starts with a 0 or a 1. If it starts with a 1, it returns true; otherwise it returns false. Give as an input to our EquivalentInput checker M', '0' and '1', and return the opposite of what the EquivalentInput returns. If $M(x)$ doesn't halt, then M' will treat '0' and '1' the same; if it does halt, it will treat '0' and '1' differently.

7. You are helping a group of astronomers observe a subset of possible important events in the sky. Events occur at times $t_1, t_2, \ldots, t_n$. They will be hiring students to observe the events. Students can be hired for different time intervals. They are given a list of options, each with an interval and a cost: student j costs c_j and can cover events in the range $[t_{s_j}, t_{f_j}]$. It is OK to have observers overlap in time. They would like to hire observers to record all events as cheaply as possible.

In the example below, assume that there are 8 events.

	student 1	student 2	student 3	student 4	student 5
interval	$[t_1, t_5]$	$[t_1, t_3]$	$[t_2, t_6]$	$[t_4, t_8]$	$[t_6, t_8]$
cost	10	2	5	4	3

Possible solutions are to hire students 1 and 5 for a cost of 10+3=13, or hire students 2, 3 and 5 for the cost of 2+5+3=10, or can hire students 2 and 4 for the cost of 2+4=6, which is the cheapest.

Give a polynomial-time algorithm that finds the set of students that can observe all events $\{t_1, t_2, \ldots, t_n\}$ as cheaply as possible. Your algorithm can return only the cost, and not the set.

*You must **explain how your algorithm works** (for example, explain the meanings of all variables other than loop counters) and **analyze its running time**, but you do not need to provide a proof of correctness.*

Solution. We will use dynamic programming. We will compute the minimum cost to cover the events $\{t_1, t_2, \ldots, t_i\}$ for all i. We'll call this cost $Opt(i)$. Let S_i denote the set of students who can cover event i.

> `Let` $Opt(0) = 0$ `For` $i = 1, \ldots, n$ $Opt(i) = \min_{j \in S_i} c_j + Opt(s_j - 1)$ `Endfor Output` $Opt(n)$`.`

Running time. The algorithm computes Opt values, for n values, and for each, there may be up to m possible students j to consider, for a total running time of $O(mn)$.

Alternate Solutions. Of course, it is just as good to set on the "backwards", with $Opt(i)$ denoting the optimal solution for times $t_i, t_{i+1}, \ldots, t_n$.

It is great if they set this up as a shortest path (Bellman-Ford or actually Dijkstra in this case) application. Nodes are the times to observe nodes, say with an artificial start node t_0, so we have $n + 1$ nodes. Will look for a path from 0 to n. Edges go backwards in time for 0 cost, and forwards in time corresponding to each student. For a student available for interval $[t_{s_j}, t_{f_j}]$ we add an edge from $s_j - 1$ to t_j at the cost of c_j. A path in this graph from 0 to some node i corresponds to a sequence of students hired that cover events $t_1, t_2, \ldots, t_i$ with backwards edges corresponding to overlapping student times.

8. For the following two subproblems provide one of the following two answers:

- There is a polynomial-time algorithm for the problem; or
- the problem is NP-complete.

Explain why your answer is correct.

(a) Given a complete directed graph on a set of n nodes V and costs $c_{ij} \geq 0$ for traveling from node i to node j for every pair of nodes, and a number $\gamma > 0$. Is there a simple cycle C through all n nodes with total cost at most γ?

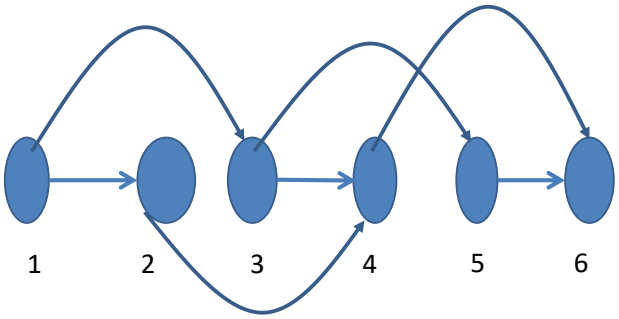
Solution Problem is NP-complete. To prove it is NP-complete, we show it is in NP: given a cycle, we can verify it goes through all the nodes, and its total cost less than γ by adding the costs. It is NP-complete, as its harder than the Hamiltonian cycle problem. For solving the HC problem by this problem, use $c_{ij} = 0$ if (i, j) is an edge, and if its not, then cycle through all the nodes with cost $\gamma = 1$ is exactly a traveling salesman tour.

(b) Given a complete directed graph on a set of n nodes V and costs $c_{ij} \geq 0$ for traveling from node i to node j for every pair of nodes, and a pair of nodes s and t, and a number $\gamma > 0$, is there a simple cycle C through edge (s, t) with cost at most γ.

Solution Solvable in polynomial time. Delete edge (i, j), and find the shortest path from j to i in the remaining graph. The cycle exists if an only if the length of the path is at most $\gamma - c_{ij}$.

9. You are helping a scientific observation team. They are hoping to observe many events. Assume events are numbered in the order of when they occur, so the times are $t_1 < t_2 < \ldots < t_n$. Each event i has an expected scientific value $v_i \geq 0$, and the goal is to observe a subset of events I with $\sum_{i \in I} v_i$ as high as possible. Unfortunately, some pairs of events cannot be both observed as taking equipment from one observation to a next takes time.

There are some complicated rules that govern what can be done right after observing the event at time t_i, depending on the equipment needed, the time and location of the event, etc. You are given a directed graph $G = (V, E)$ whose nodes correspond to the events, and edge (i, j) indicates that event $j > i$ and j can be observed right after i. Note that all edges go from earlier times to later times. You observe right away that there is no harm in observing event 1, as it occurs so much earlier that it has a directed path to all other events, i.e., observing event 1 doesn't exclude observing any other event later.



For an example consider the graph G below of 6 possible observations. Assuming all values are equal (say $v_i = 1$ for all i), the highest value solution is to observe events 1, 2, 4 and 6. It would alternately possible to observe events 1, 3, 5, and 6, but it's not possible to observe both 2 and 3 or both 4 and 5.

Give a polynomial time algorithm that finds the subset of observable events of maximum scientific value. Your algorithm can return only the value, and not the set. Your algorithm may take advantage of the observation that there is no harm in observing event 1, as noted above.

Solution 1: using shortest path We can set up the problem as a shortest path problem with the "cost" of edge $(i, j) = -v_j$. Now compute the min-cost path from 1 to all other nodes j, call this c_j, and return the $\max_j v_1 - c_j$, as the max value.

Running time. we need to use Bellman-Ford as the costs are negative, and that takes $O(mn)$ where $|E| = m$. It is also possible to do this in $O(m)$ time, as the graph is acyclic, so a shortest path to node node j must come from an entering edge (i, j) and $i < j$, so we can compute the distances in the topological order.

Correctness. By assumption observing event 1 is always part of the solution. The sequence of observable events forms a path in G, and the value of the path is v_1 minus the cost of the path.

Solution 1a The same shortest path base solution can also be explained directly if one bases it on the fact that the graph is acyclic.

Set $r_1 = v_1$
For $j = 2$ to n
    let $r_j = \max_{(ij)\in E} r_i + v_j$
return $\max_j r_j$

Running time is clearly $O(|E|)$ as setting the r_j value is proportional to its degree. Of course also accept $O(n^2)$ as that is an upper bound using n only.

Solution 2: using dynamic programming Let $Opt(i)$ denote the maximum value solution considering only the first i events, ending with event i. So clearly $Opt(1) = v_1$. To get the remaining $Opt(j)$ values, use the recurrence for $j > 1$:

$$Opt(j) = \max_{(i,j)\in E}(Opt(i) + v_j)$$

Then return the maximum possible value is $\max_i Opt(i)$.

Running time. We have n values we need to compute, and take $O(n)$ time to compute one of the values as we take the minimum of at most n options, so a total of $O(n^2)$ time. Alternately, $O(m)$ also work, where $m = |E|$, if j has d_j^+ entering edges, we spend $O(d_j^+)$ time computing $Opt(j)$ and $\sum_i d_i^+ = |E|$.

Correctness. We prove that $Opt(j)$ is the maximum value solution considering only the first j events. This is clearly true for $j = 1$. For a larger j value we use induction on j. Clearly $Opt(j) \leq Opt(i) + v_j$ for any edge (i, j) as by induction $Opt(i)$ is the optimal solution till event i, and observing j after i is an option. This proves that $Opt(j) \geq \max_{(i,j)\in E}(Opt(i) + v_j)$.

On the other hand, we know that there is no harm in observing the first event, so j is not first in the optimal solution. Let $i < j$ be the previous observation. Now (i, j) must be an edge, and we have $Opt(j) = Opt(i) + v_i$, finishing the proof.

Alternate dynamic program One can also do this backwards. So $Opt(i)$ is now the max value solution from event i to the end, still insist that it must include event i. This is trickier, as event i can be last. So we get the following recurrence.

$$Opt(i) = \max(v_i, \max_{(j,i)\in E}(Opt(j) + v_i)$$

Now we need to return $Opt(1)$ which is simpler.

10. Consider the following expert selection problem. For a movie project you need a bunch of different experts, from legal experts, through experts on the computer science company cultures, etc. You have a list of possible experts that you can hire, a set S of experts, but some of them are quite expensive. You need help with a set A of areas. Expert i needs a fee of c_i to be hired. The good news is that some of the experts have more than one expertise, though no single expert has more than two skills on your list. You would like to hire a subset I of the experts so that you have an expert on the payroll in all areas in the set A, while minimizing the total cost $\sum_{i\in I} c_i$.

Give a 2-approximation algorithm for this problem. Your algorithm should run in polynomial time, and find a set of experts that covers all required topics, and cost at most 2 times the minimum possible.

Solution Take the set i that is cheapest and covers some new element. Running time is $O(n \log n)$ with n experts, starts with sorting the fees, and keep an array of all the needed skills so we can mark the skills not yet covered.

This is a 2-approximation for the following reason: For each skill a let $i^*(a)$ be the expert providing this skill in the optimal solution, and $i(a)$ be the expert covering this skill in the greedy algorithm. Now consider the experts in order of the greedy algorithm. For each skill a, we have $c_{i(a)} \leq c_{i^*(a)}$ by the order elements are chosen. The algorithm's total cost is at most $\sum_a c_{i(a)}$ (can be less if some expert chosen covers multiple skills). If I^* denotes the optimal solution, we have

$$\sum_a c_{i(a)} \leq \sum_a c_{i^*(a)} \leq 2 \sum_{i \in I^*} c_i$$

where the last inequality follows as any expert $i \in I^*$ can cover at most 2 skills.