

Multiple Choice

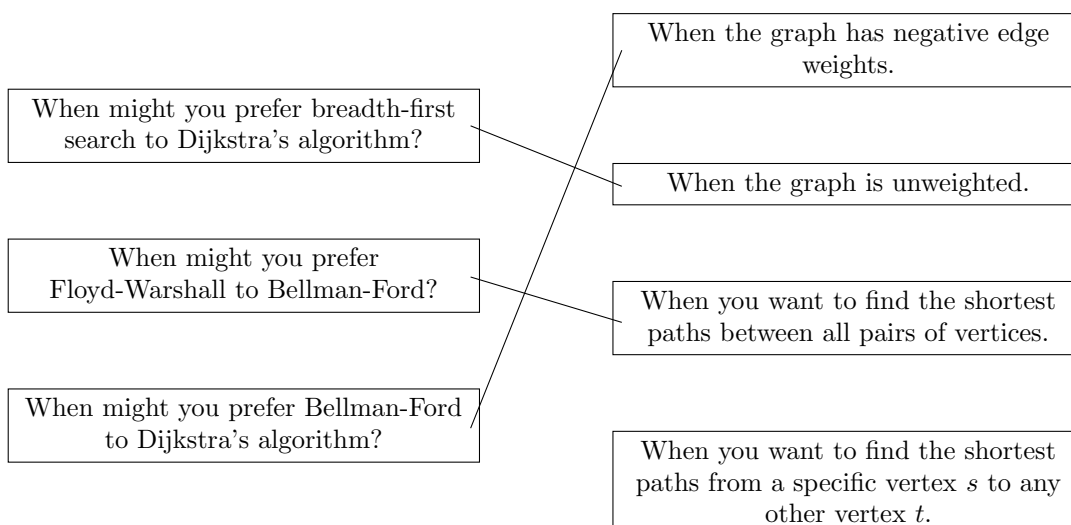
1. (a) A, C; (b) A, B, C, E; (c) D; (d) A
2. (a) D; (b) D
3. E
4. A, B, D
5. A
6. (a) A; (b) A
7. C
8. (a) A; (b) A; (c) A
9. B
10. (a) E; (b) A
11. B
12. D
13. (a) D; (b) D
14. (a) A, D, E; (b) C

Short Answers

1. All three algorithmic paradigms involve expression larger subproblems in terms of smaller subproblems. However, divide-and-conquer exhaustively explores all subproblems, including solving repeated subproblems multiple times; dynamic programming exhaustively explores all subproblems, but caches the answers to solved subproblems so that each subproblem only gets solved once; and greedy doesn't exhaustively explore all subproblems—instead it only explores the locally optimal subproblem.
2. The $\Omega(n \log(n))$ -time lower bounds *comparison-based* sorting algorithms; however, linear-time sorting algorithms leverages prior knowledge about the distribution and structure of the elements being sorted to avoid exhaustive comparisons.
3. Consider a graph G with three vertices A , B , and C with edges (A, B) , (B, C) , (A, C) with edge weights $w(A, B) = 2$, $w(B, C) = -2$, and $w(A, C) = 1$. The shortest path from A to C is via B ; however, Dijkstra's algorithm will not find this path. Adding 2 to all of the edge weights will not help either. In the modified graph G' , $w(A, B) = 4$, $w(B, C) = 0$, and $w(A, C) = 3$, and Dijkstra's algorithm will not find this path.

Dijkstra's algorithm still doesn't work because paths are unfairly penalized for the *number* of edges in the path and not evaluated solely on the weight of the edges in the path.

4. Here are the matches:



5. The value of any flow f is at most the value of any s-t cut $S, V \setminus T$. Surprisingly, there exists a flow f_{max} and s-t cut S_{min} and $T = V \setminus S_{min}$ whose values are equal!

Problems

1. (a) There are a few valid solutions.

Soln. 1 Let $T(i, 0)$ be the length of the longest zig-zag sequence ending at exactly element i with that element being greater than the previous element in the sequence. Let $T(i, 1)$ be the length of the longest zig-zag sequence ending at exactly element i with that element being less than the previous element in the sequence.

$$T(i, 0) = \max_{k < i: A[k] < A[i]} T(k, 1) + 1$$

$$T(i, 1) = \max_{k < i: A[k] > A[i]} T(k, 0) + 1$$

Soln. 2 Let $T(i, 0)$ be the length of the longest zig-zag subsequence of $A[0 : i + 1)$ with the last element in the subsequence being greater than the second-to-last element. Let $T(i, 1)$ be the length of the longest zig-zag subsubsequence in $A[0 : i + 1)$ with the last element in the subsequence being less than the second-to-last element.

This approach is tricky; in order for it to work, you need to store the value or index of the last element in the subsequence for each $T(i, 0)$ and $T(i, 1)$, denoted below as $T(i, 0).last$ and $T(i, 1).last$.

$$T(i, 0) = \max\{T(i - 1, 0), (T(i - 1, 1) + 1) \cdot \mathbb{1}[T(i - 1, 1).last < A[i]]\}$$

$$T(i, 1) = \max\{T(i - 1, 1), (T(i - 1, 0) + 1) \cdot \mathbb{1}[T(i - 1, 0).last > A[i]]\}$$

```

(b) def zigZag(A):
    n = len(A)
    DP = {}
    DP[(0, 0)] = DP[(0, 1)] = 1
    for i in range(n):
        opt0 = [DP[(k, 1)] + 1 if A[k] < A[i] else 0 for k in range(i)]
        DP[(i, 0)] = max(opt0)
        opt1 = [DP[(k, 0)] + 1 if A[k] > A[i] else 0 for k in range(i)]
        DP[(i, 1)] = max(opt1)
    # Returns the maximum value in DP
    return max(DP)

```

(c) The two approaches are backtracking or maintaining an additional data structure with the cached information.

```

2. (a) def prioritize(tasks):
    # Suppose tasks is a list of (taskId, deadline) tuples
    sortedTasks = sorted(tasks, key=lambda t: t[1]) # sorts tasks by increasing deadlines
    curTime = 0
    todo = []
    for i in range(n):
        taskId, deadline = sortedTasks[i]
        if deadline >= curTime + 1:
            todo.append(taskId)
            curTime += 1

```

(b) (i) After t tasks have been added to `todo`, there exists a feasible and optimal solution T^* such that $T^*[0, \dots, t-1] = \text{todo}[0, \dots, t-1]$.

(ii) After 0 tasks have been added to `todo`, it's an empty list. All feasible and optimal solutions must contain the empty list.

(iii) Suppose the inductive hypothesis holds for t . Here, we prove it holds for $t+1$.

Let T^* be an optimal schedule such that $T^*[0, \dots, t-1] = \text{todo}[0, \dots, t-1]$. Suppose we add task i as `todo[t]` but T^* assigns task $j \neq i$ as $T^*[t]$ such that j is the task with the earliest deadline of the tasks remaining in T^* . Exchanging task j for task i in T^* results in a new schedule T^*_{new} , and we claim this schedule is still feasible and optimal.

This schedule is still feasible since:

- Tasks 0 to $t-1$ are feasible by the inductive hypothesis.
- Task t can be finished by its deadline with tasks 0 to $t-1$ already scheduled by construction of the algorithm.
- Tasks $t+1$ to $n-1$ can be done by their deadlines given task i since the completion of task i in `todo` must be no later than the completion of task j in T^* since the greedy algorithm assigns it the earliest available time, given that t tasks have already been assigned.

This schedule is optimal since it has the same number of tasks as an optimal schedule.

Thus, T^*_{new} is feasible and optimal.

(iv) Suppose the inductive hypothesis holds for $j = t^*$ where t^* is the length of an optimal schedule. Then there exists a T^* such that $T^*[0, \dots, t^*-1] = \text{todo}[0, \dots, t^*-1]$. In other words, `todo` = T^* . Thus, the algorithm returns a feasible and optimal schedule, as desired.

```

(c) def weighted_prioritize(tasks):
    # Suppose tasks is a list of (taskId, deadline, value) tuples
    sortedTasks = sorted(tasks, key=lambda t: t[2])
    reverseSortedTasks = sortedTasks[::-1] # sorts tasks by decreasing values

    deadlines = [t[1] for t in tasks]
    timeSlots = [True for _ in range(max(deadlines))]
    todo = []
    for i in range(n):
        taskId, deadline, value = reverseSortedTasks[i]
        flooredDeadline = floor(deadline)
        j = flooredDeadline - 1
        while j >= 0:
            if timeSlots[j]:
                break
        if j >= 0:
            timeSlots[j] = False # mark timeslot as unavailable
            todo.append(taskId)

```

- (d) In words, our algorithm proceeds as follows: construct and complete a table S where $S(i, t)$ represents the value of the optimal schedule using a subset of the tasks $\{0, \dots, i-1\}$ and only using time up until t . This satisfies the recurrence:

$$S(i, t) = \begin{cases} \max\{S(i-1, t), v_i + S(i-1, t-t_i)\} & \text{if } d_i \geq t \\ S(i, t) = S(i-1, t) & \text{otherwise} \end{cases}$$

Backtrack to return the optimal schedule.