Chapter 21 Practice Exercises

Disclaimer: These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. Which of the following statements is/are **TRUE** regarding the brute force algorithmic approach?
   - **I.** A correctly implemented brute force algorithm may still fail to find the optimal solution to an optimization problem.
   - **II.** Given any problem with an input size of n, a brute force solution for that problem can never exceed a time complexity of $\Theta(2^n)$.
   - **III.** Brute force algorithms may need to rely on immense computational power to work and can take significant time to run.

   A) II only
   B) III only
   C) I and III only
   D) II and III only
   E) I, II, and III

2. Which of the following is an optimization problem?
   A) Finding the number of ways to schedule rooms for final exams without any conflicts
   B) Finding the ideal number of gadgets that should be produced at a factory to maximize profits
   C) Finding a path out of a complicated maze
   D) Finding the MST of a connected graph
   E) More than one of the above

3. A greedy approach can always be used to solve an optimization problem as long as:
   - **I.** The problem has an optimal substructure.
   - **II.** The problem has a unique solution.
   - **III.** The problem satisfies the greedy-choice property.

   A) III only
   B) I and II only
   C) I and III only
   D) II and III only
   E) I, II, and III

4. Consider the greedy solution to the coin-change problem, in which the minimal number of coins needed to return any change amount is computed by greedily selecting the highest-value coin that does not bring you over the desired amount. For which of the following coin denominations would the greedy approach *always* succeed in finding the optimal solution to this problem?
   - **I.** {1¢, 5¢, 10¢, 25¢}
   - **II.** {1¢, 8¢, 16¢, 20¢}
   - **III.** {1¢, 3¢, 9¢, 27¢}

   A) I only
   B) I and II only
   C) I and III only
   D) II and III only
   E) I, II, and III

5. Which of the following statements is/are **TRUE**?
   - **I.** If an optimization problem can be solved with a greedy approach, and the first greedy choice adds an element x to the solution, then there is still a chance that no optimal solution to the problem includes the element x.
   - **II.** If an optimal solution to an optimization problem can be constructed efficiently from optimal solutions of its subproblems, then a greedy approach is guaranteed to work on that problem.
   - **III.** Mathematical induction is a useful technique that can be used to prove the efficacy of a greedy approach when solving optimization problems.

   A) II only
   B) III only
   C) I and III only
   D) II and III only
   E) I, II, and III

6. Suppose you decided to help EECS 281 students by offering special 1-on-1 tutoring sessions on the day before the final exam. You send out a sign-up form to every EECS 281 student in the class that allows them to sign up for these tutoring sessions. On this form, each student can submit their ideal start time and the expected duration of their 1-on-1 session (e.g., a student requesting an 11 AM start time and a duration of 45 minutes would like to have a session that lasts from 11 AM to 11:45 AM). Given multiple student requests, which of the following greedy algorithms would *always* allow you to help the greatest number of students? You may assume that all requests are restricted between 8 AM and 8 PM on the day before the final exam.
   A) Greedily selecting the available student requests with the earliest desired start time
   B) Greedily selecting the available student requests with the earliest desired end time
   C) Greedily selecting the available student requests with the latest desired end time
   D) Greedily selecting the available student requests with the smallest desired duration
   E) More than one of the above

7. Suppose you are beginning a road trip in a car that can go at most m miles on a full tank. Given an *unsorted* vector of n integers representing the distances of gas stations on your route (in miles) from your starting position, what is the worst-case time complexity of identifying the minimum number of stops you can make to travel to a destination d miles away without running out of gas (assuming you start with a full tank), if you use the most efficient algorithm?
   A) $\Theta(n)$
   B) $\Theta(d+n)$
   C) $\Theta(dn)$
   D) $\Theta(n\log(n))$
   E) $\Theta(dn\log(n))$

8. You decide to use a brute force algorithm to calculate the shortest distance between two vertices in a graph. Using this strategy, you obtain a solution of 281. If your brute force algorithm is implemented correctly, which of the following can you safely assume?
   A) The optimal distance is exactly 281
   B) The optimal distance may be less than 281
   C) The optimal distance may be greater than 281
   D) All paths between the two vertices have a distance of 281
   E) We cannot safely assume any of the above statements

9. Your friend is trying to solve the coin change problem to minimize the number of coins needed to make change given a set of coin denominations. They have written two algorithms to solve the problem: one that uses brute force, and one that uses the greedy approach of always selecting the highest denomination coin that is available. Assuming that both algorithms are implemented correctly, which of the following outcomes is/are **NOT** possible when solving for the same target change amount with the same coin denominations?
   A) The brute force solution returns 281, and the greedy solution returns 280
   B) The brute force solution returns 281, and the greedy solution returns 281
   C) The brute force solution returns 281, and the greedy solution returns 282
   D) Both (A) and (C)
   E) All of (A), (B), and (C)

10. Which of the following statements is **FALSE** regarding the divide-and-conquer algorithm family?
   A) Divide and conquer algorithms often have time complexities that involve a logarithmic term (e.g., $\log(n)$)
   B) The Master Theorem can be a useful tool for analyzing the time complexity of divide-and-conquer algorithms
   C) Divide-and-conquer works best if a problem can be split up into multiple subproblems whose solutions depend on each other
   D) Divide-and-conquer algorithms can often be parallelized, where work is distributed to different processors and done simultaneously
   E) None of the above

11. Which of the following problems can be solved using divide-and-conquer?
   A) Given a one-dimensional array of both positive and negative numbers, identify the subarray of numbers that produces the largest sum
   B) Given an array of integers, sort the array in ascending order
   C) Given two integers m and n, compute the value of m^n
   D) Given multiple points on a two-dimensional plane, find the two points that are closest to each other
   E) All of the above

12. Which of the following sorting algorithms utilize a divide-and-conquer (or combine-and-conquer) approach?
   I. Mergesort
   II. Quicksort
   III. Heapsort

   A) I only
   B) II only
   C) I and II only
   D) I and III only
   E) I, II, and III

13. You are given a list of n tasks, as well as a vector `duration` where `duration[i]` is the duration of the i^{th} task in minutes. Given you have `total_time` minutes to work on the tasks, implement a function that returns the maximum number of distinct tasks that you can complete with `total_time` minutes. The tasks can be completed in any order.

```
int32_t max_tasks(const std::vector<int32_t>& duration, int32_t total_time);
```

Example: Given `duration = [1, 4, 2, 3, 1]` and `total_time = 7`, you would return 4, since you can complete all but the job with duration 4 in the given time ($1 + 2 + 3 + 1 = 7$).

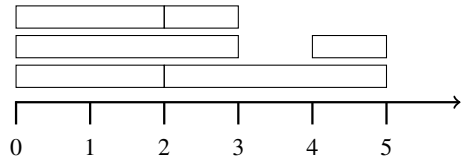
14. Consider the following `Lecture` entity, which stores the start and end time of a given lecture:

```
struct Lecture {
  int32_t begin_time;
  int32_t end_time;
};
```

You are given a vector of n `Lecture` entities. Implement a function that returns the minimum number of classrooms needed to schedule all the lectures such that no two lectures occur at the same time in the same room. Note that it is valid for two lectures to happen back-to-back (e.g., if one lecture ends at time 5 and another starts at time 5, they can be sent to the same classroom). You may assume that the begin and end times are all valid (i.e., begin time < end time).

```
int32_t min_classrooms_needed(const std::vector<Lecture>& lectures);
```

Example: Given lectures with the following beginning and ending times: [0, 2], [0, 2], [0, 3], [2, 3], [2, 5], [4, 5], you would return 3, since it is possible to schedule all these lectures using just 3 classrooms. One potential arrangement is shown below:



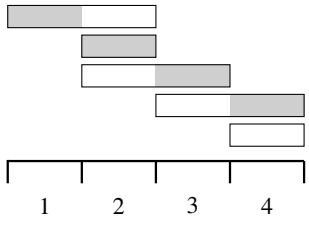
15. You are attending a multi-day programming conference where many guest lecturers are scheduled to give presentations. Consider the following `Presentation` entity, which stores the start and end dates of a given presentation:

```
struct Presentation {
  int32_t begin_date;
  int32_t end_date;
};
```

In this case, a lecturer will give the same presentation on all days between the begin date and end date, inclusive. For example, if begin date is 2 and end date is 4 for a given presentation, that means the presentation will be given on days 2, 3, and 4. You may only attend one presentation on any given day. Given a vector of `Presentation` entities, implement a function that returns the maximum number of distinct presentations that you will be able to attend at the conference.

```
int32_t max_presentations(const std::vector<Presentation>& presentations);
```

Example: Given presentations with the following beginning and ending dates: [1, 2], [2, 2], [2, 3], [3, 4], [4, 4], you would return 4, since it is possible for you to attend at most 4 of these presentations (the shaded section represents the day you attend a presentation):



Chapter 21 Exercise Solutions

1. **The correct answer is (C).** A brute force algorithm, when implemented correctly, would check all possible solutions before deciding on the solution that solves the problem. This means that a brute force approach can ensure that the optimal solution is found, so statement I is false. Statement II is not guaranteed to be true either depending on the problem to be solved — if the number of possible solutions exceeds 2^n, then the time complexity of brute force can exceed $\Theta(2^n)$ (one example is the Traveling Salesperson Problem (TSP), which has a brute force time complexity of $\Theta(n!)$, which we will discuss in more detail in the following chapter). Statement III is true: since brute force needs to check the entire solution space, it is computationally intensive and may take significant time to run (and if a machine is not powerful enough, a brute force algorithm may end up taking an infeasible amount of time to complete).

2. **The correct answer is (E).** Both (B) and (D) are optimization problems: (B) seeks to find the maximum profits under a set of constraints, while (D) seeks to find the minimum total weight required to connect all vertices in a graph.

3. **The correct answer is (C).** If a problem has an optimal substructure and satisfies the greedy-choice property, we can use mathematical induction to prove the correctness of a greedy approach.

4. **The correct answer is (C).** The denominations in I are the just the standard coin denominations in real life, which we proved can be solved using a greedy approach in example 21.4. The denominations in II do not work for 24¢, since the greedy solution would return 4 coins (20¢, 1¢, 1¢, 1¢) when the optimal solution is 3 coins (8¢, 8¢, 8¢). We can prove that III is optimal using the same exchange argument that we used to prove I: we know the number of 1¢ coins in our solution must be less than 3, since otherwise we can replace these coins with a single 3¢ coin. We also know that the number of 3¢ and 9¢ coins are each less than 3 as well, for the same reason. Using a proof by cases, we can show that:
   - If the amount of change is greater than 27¢, then the greedy choice adds this coin to the solution. If this is not optimal, there must be a way to return a change amount above 27¢ without a 27¢ coin. This involves at least 3 coins worth of 9¢, 3¢, or 1¢ coins, which would not be optimal.
   - If the amount of change is between 9¢ and 26¢, then the greedy choice would be to add a 9¢ coin to our solution. If this is not optimal, there must be a way to return this change amount without a 9¢ coin. This would require at least 3 coins worth of 3¢ or 1¢ coins, which would not be optimal.
   - If the amount of change is between 3¢ and 9¢, then the greedy choice would be to add a 3¢ coin to our solution. If this is not optimal, there must be a way to return this change amount without a 3¢ coin. This would require at least 3 coins worth of 1¢ coins, which would not be optimal.
   - If the amount of change is less than 3¢, we only have one type of coin that can be used, which trivially implies that the optimal solution must contain this coin.

5. **The correct answer is (B).** Statement I is false because a greedy approach does not look back and undo any choices, so if a greedy approach is known to work and x is added to the solution, there must be at least one optimal solution that includes x. Statement II is false because an optimal substructure alone cannot prove that greedy always works: we also need to show that the greedy-choice property holds, and that every greedy choice we make has the potential to be a valid solution. Statement III is true, since mathematical induction can be used to show that a greedy approach is valid after identifying that the greedy-choice property and an optimal substructure holds for a given problem.

6. **The correct answer is (B).** This is a variation of the activity selection problem introduced in section 21.3.3. The correct greedy approach for this type of problem is to select the request with the earliest end time.

7. **The correct answer is (D).** This is the breakpoint selection problem introduced in section 21.3.4: to solve this problem, we sort the gas stations in ascending order and greedily select the last gas station we can reach before running out of gas. The bottleneck of this algorithm is the sorting step, which takes $\Theta(n \log(n))$ time.

8. **The correct answer is (A).** A brute force solution examines all possible solutions, so if returns 281 as the solution, then 281 must be the optimal solution (otherwise it would have discovered a solution better than 281).

9. **The correct answer is (A).** The brute force approach examines all possible solutions, so it is guaranteed to return the best solution if implemented correctly. Thus, it is not possible for a greedy approach to do better, so choice (A) is not possible.

10. **The correct answer is (C).** Divide-and-conquer works best when a problem can be split into subproblems that are independent from each other, which allows the input to be broken up, solved individually, and combined together without having to worry about dependencies between subproblems. If there is a dependency between different subproblems, then another algorithmic approach my work better (e.g., dynamic programming, which will be discussed in a later chapter).

11. **The correct answer is (E).** All of the above can be solved using divide-and-conquer. Option (A) was demonstrated in section 21.4.4, option (B) can be solved using a divide-and-conquer sorting algorithm like mergesort or quicksort, option (C) was demonstrated in section 21.4.2, and option (D) was briefly mentioned at the end of the chapter (and will be covered in section 26.3).

12. **The correct answer is (C).** Mergesort and quicksort are two divide-and-conquer sorting algorithms, since they break up the input into smaller subproblems, solve those subproblems independently, and then combine the subproblems to obtain the final solution (i.e., a fully sorted array).

13. The greedy solution to this problem is to sort all the tasks in order from least to most time required, and then greedily select the tasks that require the least amount of time. The proof for this is pretty intuitive — if somehow the optimal solution does not include the task that takes the least amount of time, we can obtain a solution that is no worse by swapping out a longer task with the one that takes the least amount of time (a contradiction). One possible solution is implemented below:

```
int32_t max_tasks(const std::vector<int32_t>& duration, int32_t total_time) {
  int32_t num_tasks = 0;
  std::vector<int32_t> sorted_durations(duration);
  std::sort(sorted_durations.begin(), sorted_durations.end());
  for (int32_t curr_duration : sorted_durations) {
    if (curr_duration > total_time) {
      break;
    } // if
    total_time -= curr_duration;
    ++num_tasks;
  } // for i
  return num_tasks;
} // max_tasks()
```

14. The greedy solution to this problem is to sort all the lectures in order of starting time, and then iterate over the lectures and assign them to an available classroom (or allocate a new one if all the existing classrooms are full). This is guaranteed to identify the maximum number of lectures in conflict at any given time, since a room $d+1$ will only be assigned under this approach if d other classrooms are occupied. Note that if the maximum number of lectures that conflict with each other at any given time is d, then all possible solutions to the problem must require at least d classrooms (since anything less than d would cause two lectures to overlap), which makes d the optimal solution. One possible implementation is shown below: here, we use a priority queue to keep track of the end times of all the lectures at any point in time. Whenever we want to add a new lecture, we remove any lectures that may have completed and add the new lecture in, keeping track of the largest size the priority queue was able to reach.

```
struct LectureCompare {
  bool operator() (const Lecture& lhs, const Lecture& rhs) {
    return lhs.begin_time < rhs.begin_time;
  } // operator()()
};

int32_t min_classrooms_needed(const std::vector<Lecture>& lectures) {
  std::vector<Lecture> sorted_lectures(lectures);
  LectureCompare comp;
  std::sort(sorted_lectures.begin(), sorted_lectures.end(), comp);

  int32_t min_classrooms = 0;
  std::priority_queue<int32_t, std::vector<int32_t>, std::greater<int32_t>> pq;
  for (const Lecture& lecture : sorted_lectures) {
    while (!pq.empty() && pq.top() <= lecture.begin_time) {
      pq.pop();
    } // while
    pq.push(lecture.end_time);
    min_classrooms = std::max(min_classrooms, static_cast<int32_t>(pq.size()));
  } // for

  return min_classrooms;
} // min_classrooms_needed()
```

15. The greedy solution to this problem is to sort all the presentations in order of starting time, and then iterate over the presentations to identify which one you should attend for each day. If there are multiple presentations that you can attend on a single day, you would greedily select the presentation with the earliest end date. Why does this work? Suppose there are two presentations that you attend on any given day, denoted as P_1 and P_2, where P_1 ends before P_2. Let's say that the optimal solution is to select P_2 instead of P_1 on this date. However, in this case, we can always construct another schedule that chooses P_1 instead of P_2 that is no worse than the schedule that chose P_2.
    - If P_1 can be chosen on some later day, we can always swap P_1 and P_2 and still be able to attend both presentations (since if we can choose P_1, we can also choose P_2 since we know it ends later).
    - If P_1 cannot be chosen on some later day, we can replace P_2 with P_1 when making our choice and still end up with a solution that is no worse than the outcome we obtained from making the opposite choice.

    Thus, it is optimal to always choose the presentation that ends sooner, if there are multiple presentations that you can attend on a single day. The implementation to this problem can be done very similarly to the previous problem: we sort the presentations in order of start time, and then for each day you can attend a presentation, we push in the available presentation end times into a min-priority queue and then choose the available presentation at the top of the priority queue. An implementation is shown below:

```
struct PresentationCompare {
  bool operator() (const Presentation& lhs, const Presentation& rhs) {
    return lhs.begin_date < rhs.begin_date;
  } // operator()()
};

int32_t max_presentations(const std::vector<Presentation>& presentations) {
  std::vector<Presentation> sorted_presentations(presentations);
  PresentationCompare comp;
  std::sort(sorted_presentations.begin(), sorted_presentations.end(), comp);
  std::priority_queue<int32_t, std::vector<int32_t>, std::greater<int32_t>> pq;

  int32_t max_day = 0;
  for (const Presentation& presentation : sorted_presentations) {
    max_day = std::max(max_day, presentation.end_date);
  } // for

  size_t idx = 0;
  int32_t solution = 0;
  for (int32_t day = 1; day <= max_day; ++day) {
    while (!pq.empty() && pq.top() < day) {
      pq.pop();
    } // while
    while (idx < sorted_presentations.size() && sorted_presentations[idx].begin_date == day) {
      pq.push(sorted_presentations[idx].end_date);
      ++idx;
    } // while
    if (!pq.empty()) {
      pq.pop();
      ++solution;
    } // if
  } // for day

  return solution;
} // max_presentations()
```