

CS106B Practice Midterm 2 (previously: Winter 2026)

Name:	SUID (numeric):
-------	-----------------

Instructions:

- **Stay within designated answer boxes**, or graders may not see your response on Gradescope. If you require extra space, write “continued” in the answer box, and then use designated scratch space somewhere in the exam. **No additional scratch paper.**
- **The time limit is strict.** When time is called, immediately drop your pencil/pen, and proceed to turn in your exam. No writing.
- For coding problems, **you may assume #include and function prototypes** already exist for you as needed.
- **Stanford Honor Code:** The Honor Code is an undertaking of the Stanford academic community, individually and collectively. Its purpose is to uphold a culture of academic honesty. Students will support this culture of academic honesty by neither giving nor accepting unpermitted academic aid on this examination.
- **Permitted and unpermitted aid:** You may have one double-sided, 8.5” × 11” page of notes. No other aids. **No electronics including laptops, phones, smart glasses/watches or earbuds within reach or sight.** Other than course staff, no consulting with other humans or AI in any way during the exam period.
- **Proctoring:** Proctors will be present in the exam room. If you have any questions or concerns during this exam, please speak with a proctor. At some point during the exam, proctors may take a few broad photos of the room, for later seating chart verification. Backpacks and phones must be left at the front of the room or with proctors. Proctors will verify your photo ID when you turn in your exam.
- If you need a **break, (e.g., bathroom)**, first give your exam to a proctor, who will log your break.

Please sign before you begin: I agree to abide by the spirit and letter of the Honor Code, and to follow the instructions above.

Signature:

1. Recursion: Dorm Struggles (15 points)

Your dorm has a tradition of going to Stanford games together and holding up signs that spell out encouraging messages, where each person is holding a sign with one letter painted on it. For example, if 10 people are going to the game, they might bring 10 signs, spelling “GOSTANFORD.” Unfortunately, some of the people in the dorm are flakes, and when they don’t show up, the remaining people are left scrambling to figure out what, if any, valid English words can be spelled with only some of the planned letters. For example, if 2 of the 10 people who had planned “GOSTANFORD” flake, they could do “TORNADOS” (10 - 2 = 8 letters) instead. Or, under more dire circumstances, “FROGS” (5 flakes, 5 letters), “ODOR” (6 flakes, 4 letters), or “OOF” (7 flakes, 3 letters). You’ve decided to write a function to assist in these circumstances:

```
Set<string> signRescue(string plan, int nFlakes);
```

Instructions and hints:

- **plan** is the message that had been planned, so its length is the number of people who originally planned to attend. For simplicity, you may assume it contains only uppercase letters (no spaces, punctuation, etc).
- **nFlakes** is the number of people who did not show up. For simplicity, you may assume that `nFlakes <= plan.length()`.
- **Return a set of all the allowed strings** that can be made with length that is exactly the remaining number of people. So for our “GOSTANFORD” example, if `nFlakes = 6`, include “ODOR” (assuming `isAllowed("ODOR")` is `true`) and not “OOF” (too short—one of the people who did show up wouldn’t get to hold a sign).
- To test whether a string would be allowed as a new sign possibility (i.e., detect English words), assume you have access to a function **`bool isAllowed(string candidate)`** that returns `true` if `candidate` would be allowed for a sign, and `false` otherwise. For simplicity, you may assume that `isAllowed()` accepts input that is in all uppercase letters.
- You may want to implement a recursive helper function for `signRescue()`. In that case, write both functions in the provided box.
- Your solution must not use any global variables, and must be fundamentally recursive in approach.
- There is no specific Big-O requirement, but excessively repetitious execution may lose points.

(scratch space—problem 1 continues on next page)

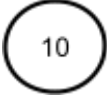
```
Set<string> signRescue(string plan, int nFlakes) {
```

```
}
```

(end of problem 1)

2. Priority Queue (10 points)

We have implemented a Priority Queue ADT as a **min-heap**, just like we did in lecture. But for ease, each element in the priority queue has just the `int` priority, not a separate `string` value. Draw a diagram of the tree shape of the **min-heap** after enqueueing the following priorities, in the order given: **10, 13, 8, 14, 3, 7**.

Diagram after inserting 10:  <i>(this one is drawn for you as a format example)</i>	Diagram after inserting 13:
Diagram after inserting 8:	Diagram after inserting 14:
Diagram after inserting 3:	Diagram after inserting 7:

(end of problem 2)

3. BTS Dorm Trip (22 points)

Recall that for the first midterm, you helped write ticket sales software for the upcoming BTS concert in Stanford stadium. Here is a copy of the relevant parts of that problem's instructions, to refresh your memory:

You've been asked to help write software to manage the re-sale website, where ticketholders can set a price to sell their tickets (like StubHub). The layout of seating in the stadium has been simplified to a rectangular shape, and will be stored in a `Grid<int> prices`, where the rows of the Grid are rows of seats starting at the stage, and the number of columns is the number of seats per row. Each value in the Grid is the ticketholder of that seat's asking price (in dollars; a non-negative value; zero is offering the ticket for free), or the constant `NOT_FOR_SALE`, which is defined for you at the top of your file as `const int NOT_FOR_SALE = INT_MAX;`. Here is an example Grid (not the actual size, so use the usual Grid class functions to find the dimensions of the Grid you are given):

	0	1	2	3
0	NOT_FOR_SALE	NOT_FOR_SALE	NOT_FOR_SALE	600
1	500	450	NOT_FOR_SALE	NOT_FOR_SALE
2	150	NOT_FOR_SALE	125	300
3	NOT_FOR_SALE	100	130	95
4	35	99	NOT_FOR_SALE	NOT_FOR_SALE

`prices:`
(Example)

*Dashed lines added,
and some seat prices
edited from midterm 1.*

The ticket website has decided to institute a new policy that **they will not allow a group of adjacent available seats to be broken up by selling only part of it**—customers must buy the entire group or none of it. They consider seats adjacent if they are **directly north, east, south, or west** from other seat(s) in the group (diagonal does not count).

Example: The above `prices` has 3 groups of adjacent available seats: `{r0c3}`, `{r1c0, r1c1, r2c0}`, `{r2c2, r2c3, r3c1, r3c2, r3c3, r4c0, r4c1}`. (Indicated by the dashed lines.)

For this midterm, you are going to implement this new policy, and also turn the ticketing system into a C++ class.

- In part (a), you will begin the class implementation.
- In part (b), you will write a breadth-first search function that identifies groups of adjacent available seats.
- In part (c), you will write a backtracking recursion function to help your dorm buy *exactly* `N` tickets, and stay within a total cost `budget`, so your dorm can go to BTS.

(scratch space—problem 3 continues on next page)

Below is the `tickets.h` file, showing our planned implementation of the ticketing system as a C++ class.

You will **not need to implement** the functions marked `n/a`, but imagine someone else is writing them. Here is what they do:

- The constructor loads the current Grid state from a backup file.
- The private helper function `computeAdjacent()` is for you to use in part (b). It returns seats to the north, east, south, and west of `loc` that are not out of bounds.
- The private helper function `totalCost` is for you to use in part (c). It returns the total cost of the provided `seats` (it does not modify the input).

```
/* tickets.h */
#include "grid.h"
const int NOT_FOR_SALE = INT_MAX;

class Tickets {
public:
    Tickets();                                // n/a
    bool isForSale(GridLocation seat);
    Set<GridLocation> gatherGroup(GridLocation seat);
    bool findNSeatsInBudget(Vector<Set<GridLocation>>& groups,
                           int N, int budget, Set<GridLocation>& result);

private:
    Set<GridLocation> computeAdjacent(GridLocation loc);    // n/a (call in (b))
    int totalCost(Set<GridLocation>& seats);                // n/a (call in (c))

    Grid<int> _prices;
};
```

(a) (2pts) Implement this function that returns `true` if the given seat is currently for sale, otherwise `false`.

```
bool Tickets::isForSale(GridLocation seat) {

}
```

(scratch space—problem 3 continues on next page)

(b) (8pts) Write a function that identifies groups of seats. It takes a `GridLocation seat`, and uses **breadth-first search (BFS)** to identify the group of adjacent available seats that includes seat. Groups of adjacent seats are, as described above, directly north, east, south, or west from other seat(s) in the group (diagonal does not count). We have written some of the function for you below, and you will just fill in the blanks. **Each blank should be only one line (or partial line) of code.**

- You may assume that the provided `seat` is available for sale. Note that your **return value should always include the given seat**.
- **Modify `_prices` to keep track of the BFS' visited** locations by setting them to `NOT_FOR_SALE`. (We don't want to include already-visited seats or seats that are not available for sale, so it works for both.)
- Use the private helper function `computeAdjacent()`, described above.
- **Example:** Given `_prices` example above and `seat = r1c0`, you should return `{r1c0, r1c1, r2c0}`. There are more available seats across the diagonal between `r1c1` and `r2c2`, but diagonal seats are not considered adjacent.
- **Example:** Given `_prices` above and `seat = r3c3`, you should return `{r2c2, r2c3, r3c1, r3c2, r3c3, r4c0, r4c1}`.

```
Set<GridLocation> Tickets::gatherGroup(GridLocation seat) {
    Set<GridLocation> result;
    Queue<GridLocation> toExplore = {seat};
    while (  ) {
        GridLocation next =  ;
        if (_prices[next] != NOT_FOR_SALE) { // available and not already visited
            _prices[next] =  ;
            result.add(next);
            Set<GridLocation> adjacent = computeAdjacent(next);
            for (GridLocation loc : adjacent) {
                 ;
            }
        }
    }
    return result;
}
```

(scratch space—problem 3 continues on next page)

(c) (12pts) Write the `findNSeatsInBudget` function to help your dorm buy *exactly* N tickets, and stay within a total cost `budget`.

- Your solution **must use backtracking-style recursion**. You must use the provided function signature directly—**no separate recursive helper function allowed** this time.
- You may assume that N and `budget` are non-negative, and that `result` is initially empty.
- You may modify `groups` if you wish.
- Return `true` if you find a set of seats that meets this criteria (and store it in `result`), otherwise `false`.
 - The `result` seats must have size exactly N.
 - The total cost of the `result` seats cannot exceed the provided `budget`.
 - The `result` seats **must obey the website's restriction** that groups of adjacent available seats cannot be sold separately (according to the definition of adjacent seats given above).
 - In the case of returning `false`, the contents of `result` don't matter.
- Assume we have already used your `gatherGroup` helper function from part (a) to identify all the groups of adjacent available seats in `_prices`, and provide them for you in the `groups` parameter.
 - **Example:** For the `_prices` example above, `groups` would contain the following 3 sets: `{r0c3}`, `{r1c0, r1c1, r2c0}`, `{r2c2, r2c3, r3c1, r3c2, r3c3, r4c0, r4c1}`.
 - So to obey the website's restriction, for each group in `groups`, you must include the entire group of seats in `result`, or none of it. (*This series of include/exclude decisions is also a hint about how to structure your recursion!*)
- Use the private helper function `totalCost()`, described above.
- **Example:** For N = 9, and the `_prices` example above, return `false`. No combination of groups of size 1, 3, and 7 will get you a set of exactly 9 seats.
- **Example:** For N = 4, `budget` = 10000, and the `_prices` example above, return `true` with `result` = `{r0c3, r1c0, r1c1, r2c0}`.
- **Efficiency:** There may be multiple correct answers for `result`, but as soon as you find one that meets the criteria, you should stop and return with it. Additionally, your function should prune its search so it does not continue adding more seats to a candidate selection of seats that *already* has more than N seats or is over `budget` (this should be handled in the base cases, not arm's-length recursion).
- *If you run out of space in the box on the next page, you may continue in the scratch space at the end of the exam.*

(scratch space—problem 3 continues on next page)

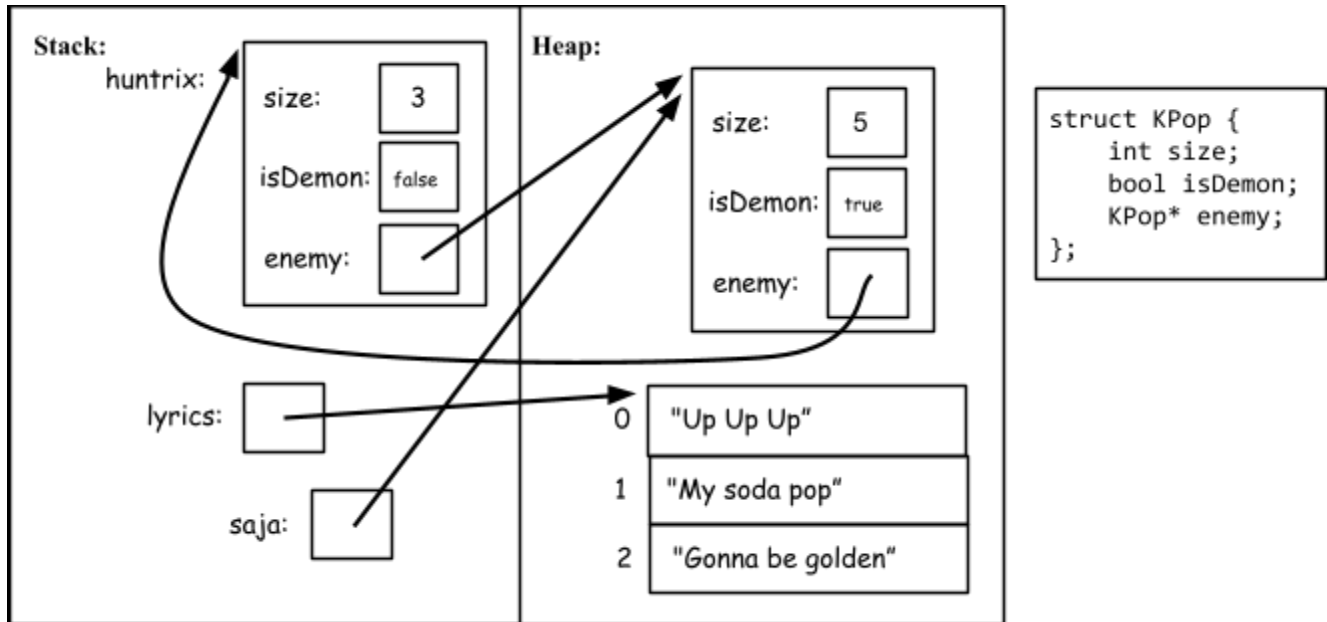
```
bool Tickets::findNSeatsInBudget(Vector<Set<GridLocation>>& groups, int N,  
                                int budget, Set<GridLocation>& result) {
```

```
}
```

(end of problem 3)

4. Memory Diagram (13 points)

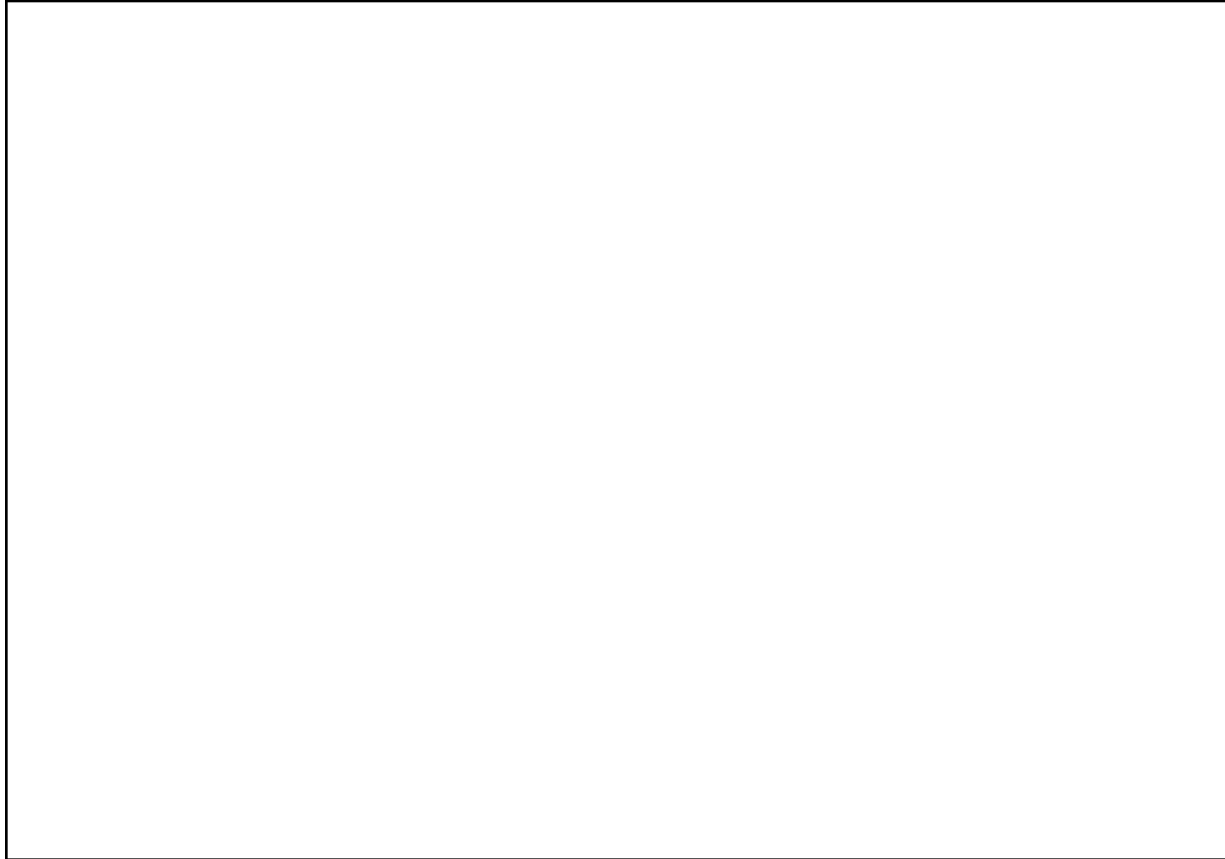
Your section leader prepared a memory diagram sample problem for your exam review. The problem consisted of a `struct` definition and a passage of code, and the solution was a memory diagram for that code (diagram shown below). Your SL would like to use this sample problem again next quarter, but unfortunately they lost the code passage. All they remember about the code passage are the first line and the second to last line, which are given in part (a). Please help your SL!



(scratch space–problem 3 continues on next page)

- (a) (11pts) Below are the first and second to last lines of the code passage. **Fill in the missing parts of the code** so that the provided memory diagram matches the result of executing the code. *Grading note:* With memory and pointers, so much depends on details such as the correct usage of *, [], ->, & and ., so we will be paying close attention to syntax for this problem.

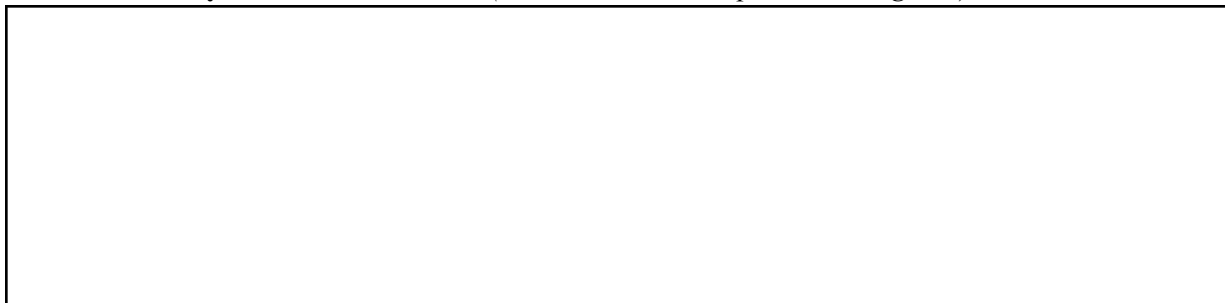
KPop huntrix; // first line



KPop* saja; // second to last line--only write one more line of code after this!



- (b) (2 pts) To improve the sample problem for next quarter, you suggest to your SL to add code to the end of the passage to clean up all the dynamically allocated memory, so there are no memory leaks. **Write the necessary code addition below.** (*You do not need to update the diagram.*)



(end of problem 4)

(scratch space—end of exam)