CPSC 221: Algorithms and Data Structures
Midterm Exam, 2016 February 23

SAMPLE SOLUTIONS

1 The O and the Omega[1] [10 marks]

In this problem, we will consider the asymptotic behaviour of functions as n goes to infinity. Each row in the table below specifies two functions $f(n)$ and $g(n)$. Fill in the **LETTER** from this list that best describes their relationship:

A. $f(n) \in \Theta(g(n))$

B. $f(n) \notin O(g(n))$, and $f(n) \notin \Omega(g(n))$

C. $f(n) \in \Omega(g(n))$, but $f(n) \notin O(g(n))$

D. $f(n) \in O(g(n))$, but $f(n) \notin \Omega(g(n))$

(Scoring will be 2 pts per correct answer, 1 pt for **blank** answers, and 0 pts for wrong answers, so random guessing isn't expected to be helpful unless you are more than 50% confident in your answer.)

We have done the first one for you, as an example.

$f(n) = \ldots$	compared to	$g(n) = \ldots$
$f(n) = n$	**D**	$g(n) = n!$
$f(n) = 3n^2$		$g(n) = 2n^3$
$f(n) = n\sqrt{n}$		$g(n) = n \lg n$
$f(n) = 1.1^n$		$g(n) = 3n^3 + 2n + 1$
$f(n) = 8n \log(n) + 4n^3 + 17$		$g(n) = 3n^3 + 8n^2 + 23 \log(n)$
$f(n) = 2^{n^2}$		$g(n) = n!$

Solution : As we warned people on Piazza, we had multiple versions of this exam: the different exam versions had different labels on this question.

If your exam has **A** printed in the example, your correct answers were ABBDB.

[1] Feel free to ignore the problems' names!

If your exam has **B** printed in the example, your correct answers were BAACA.

If your exam has **C** printed in the example, your correct answers were CDDBD.

If your exam has **D** printed in the example, your correct answers were DCCAC.

This problem was meant to be fast and easy, and fortunately almost all of you got full marks or nearly full marks. (The last question required a bit more math and that tripped up a few of you. An easy way to see this is to notice that $2^{n^2} = (2^n)^n$. I think this question may have come up on Piazza.)

A few people did poorly on this question, including all the way down to 0 points (whereas leaving it blank was worth 5 pts). We couldn't always tell whether this was a fundamental misunderstanding, really bad random guessing, or failed attempts at cheating, but in most cases, it looked like just really bad guessing. (We'll be keeping a closer eye on a few of you, though...) If you didn't get at least 8/10 on this question, make absolutely sure you understand what you did wrong and correct your understanding!

2 Recurrent Recurrences [10 marks]

Give tight big-Θ bounds for the following recurrence relations. You do not need to show your work (but it might help with partial credit). You may assume integer division (rounding down), or normal mathematical division, whichever makes your answer easier.

- Recurrence 1:

$$\begin{aligned} T(n) &= 1 \qquad \text{if } n \leq 1 \\ T(n) &= T(n/3) + 1 \qquad \text{for all } n > 1 \end{aligned}$$

Solution :

$$\begin{aligned} T(n) &= T(n/3) + 1 \\ &= T(n/9) + 1 + 1 \\ &= T(n/27) + 1 + 1 + 1 \\ &\vdots \\ &= T(n/3^k) + k \\ &\quad \text{Let } k = \log_3 n. \\ &= T(n/n) + \log_3 n \\ &= 1 + \log_3 n \\ &\in \Theta(\log n) \end{aligned}$$

Ideally, you all could recognize this recurrence, or do the unrolling in your head, and get the $\Theta(\log n)$ instantly. But if not, the unrolling should have been straightforward. If you had difficulty on this, be sure to figure out what you're not getting — seek out office hours, for example.

- Recurrence 2:

$$\begin{aligned} T(n) &= 1 \qquad \text{if } n \leq 1 \\ T(n) &= T(n/3) + n \qquad \text{for all } n > 1 \end{aligned}$$

Solution :

$$
\begin{aligned}
T(n) &= T(n/3) + n \\
&= T(n/9) + n/3 + n \\
&= T(n/27) + n/9 + n/3 + n \\
&\vdots \\
&= T(n/3^k) + n\left(\sum_{i=0}^{k-1} 1/3^i\right) \\
&= T(n/3^k) + n\left(\frac{(1/3)^k - 1}{(1/3) - 1}\right) \\
&\quad \text{Let } k = \log_3 n. \\
&= T(n/n) + n\left(\frac{(1/n) - 1}{(1/3) - 1}\right) \\
&= 1 + \left(\frac{1 - n}{(1/3) - 1}\right) \\
&= 1 - (3/2) + (3/2)n \\
&\in \Theta(n)
\end{aligned}
$$

Again, this form of recurrence is very standard, so many of you would have recognized it quickly. If not, the algebra is a bit harder, but similar to what you've seen in class and done on the assignment. (BTW, recognizing geometric series, and knowing that they converge, is a useful tidbit to remember.)

- Recurrence 3:

$$
\begin{aligned}
T(n) &= 1 \qquad \text{if } n \leq 1 \\
T(n) &= 4T(n/3) + n \qquad \text{for all } n > 1
\end{aligned}
$$

Solution :

$$
\begin{aligned}
T(n) &= 4T(n/3) + n \\
&= 4(4T(n/9) + (n/3)) + n \\
&= 16T(n/9) + (4n/3) + n \\
&= 16(4T(n/27) + (n/9)) + (4n/3) + n \\
&= 64T(n/27) + (4n/3)^2 + (4n/3) + n \\
&\vdots \\
&= 4^k T(n/3^k) + n\left(\sum_{i=0}^{k-1}(4/3)^i\right) \\
&= 4^k T(n/3^k) + n\left(\frac{(4/3)^k - 1}{(4/3) - 1}\right) \\
&= 4^k T(n/3^k) + 3n(4/3)^k - 3n \\
&\quad \text{Let } k = \log_3 n. \\
&= 4^{\log_3 n} + 3n(4/3)^{\log_3 n} - 3n \\
&= 4^{\log_3 n} + 3(4)^{\log_3 n} - 3n \\
&\in \Theta(n^{\log_3 4})
\end{aligned}
$$

The exact same solution technique works, but this one had the hardest algebra. However... this is actually *very* similar to, but much easier than, the problem you did on the assignment (where you analyzed runtime of some code and got $T(n) = 4T(n/3) + n^2$ instead). So, this was a bit of a reward for anyone who did the assignment correctly, or took the time to understand the solutions we posted.

Overall, people did fairly well on this question (the 3 parts), with most people getting roughly 7 or 8 out of 10 points, which is a pretty good sign.

3 Big-Θ Proof [10 marks]

Prove that $n + \cos(\pi n) \in \Theta(n)$. You must give a formal proof, based on the formal definition of big-Θ. (If you're rusty with your trigonometry functions, don't panic. All you need to know is that $-1 \leq \cos(x) \leq 1$ for all x. It's also handy for the next question to recall that $\cos(0) = \cos(2\pi) = \cos(4\pi) = \cos(6\pi) = \ldots = 1$, and that $\cos(\pi) = \cos(3\pi) = \cos(5\pi) = \cos(7\pi) = \ldots = -1$.)

Solution : This question ended up being marked extremely generously. There were many responses where I would have taken off many points for making a logically unsound proof, but the TAs were being more generous, due to widespread confusion over how to write a proof backwards (correctly or incorrectly). If you're still unclear about how to write a proof, please check the Piazza threads on this topic, and see a prof or TA during office hours if you're still confused. We don't promise to be equally generous when marking the final!

As to the problem itself, this was very similar to the ceiling (or was it a floor) function big-Θ proof on one of the recent sample midterms, for which we provided complete solutions.

Anyway, here's a sample solution:

Proof:

To prove that $n + \cos(\pi n) \in \Theta(n)$, by the definition of big-Θ, we need to prove that $n + \cos(\pi n) \in O(n)$ and $n + \cos(\pi n) \in \Omega(n)$.

Proof that $n + \cos(\pi n) \in O(n)$*:*
By the definition of big-O, we must find constants $c > 0$ and n_0 such that $n + \cos(\pi n) \leq cn$ for all $n > n_0$.
Consider $c = 2$ and $n_0 = 1$. Then

$$\begin{aligned} n + \cos(\pi n) &\leq n + 1 && \text{Because } \cos(x) \leq 1 \\ &< n + n && \text{Because } n > n_0 = 1 \\ &= 2n \\ &= cn \end{aligned}$$

Proof that $n + \cos(\pi n) \in \Omega(n)$*:*
By the definition of big-Ω, it suffices to prove that $n \in O(n + \cos(\pi n))$. By the definition of big-O, we must find constants $c > 0$ and n_0 such that $n \leq c(n + \cos(\pi n))$ for all $n > n_0$.
Consider $c = 2$ and $n_0 = 3$. Then

$$\begin{aligned} n &\leq n + 1 \\ &\leq n + n - 2 && \text{Because } n > n_0 = 3, \text{ so } n - 2 > 1 \\ &= 2(n - 1) \\ &\leq 2(n + \cos(\pi n)) && \text{Because } \cos(x) \geq -1 \\ &= c(n + \cos(\pi n)) \end{aligned}$$

Since $n + \cos(\pi n) \in O(n)$ and $n + \cos(\pi n) \in \Omega(n)$, we can conclude that $n + \cos(\pi n) \in \Theta(n)$. QED

Notice that our proof tells you where the proof is going (by stating what is being proven, and why), but then at each stage, the reasoning is always *forward*, starting from thing we know or are allowed to choose (like the values of c and n_0), and then deriving from that the thing we are trying to prove. This is formally the way a proof *must* be structured. (BTW, note that for the big-Ω proof, you need $n_0 \geq 2$, because when $n = 1$, the function evaluates to 0.)

Many of you prefer to write your proofs backwards. The good reason to do this is because it's often easier to devise a proof when you keep your goal clearly in mind. However, it's absolutely essential that you always remember that all of your reasoning has to work correctly going backwards: the statements you "derive" have to imply the things above them that you are trying to prove! Here's an example of a correctly written, backwards-style proof of the big-O part:
Proof that $n + \cos(\pi n) \in O(n)$*:*
By the definition of big-O, we must find constants $c > 0$ and n_0 such that $n + \cos(\pi n) \leq cn$ for all $n > n_0$.

$$\begin{aligned} & n + \cos(\pi n) \leq cn \\ \Leftrightarrow\ & 1 + \frac{\cos(\pi n)}{n} \leq c && \text{Dividing both sides by } n, \text{ requires } n > 0. \\ \Leftarrow\ & 1 + 1 \leq c && \text{If } n \geq 1, \text{ since } \cos(x) \leq 1 \\ \Leftrightarrow\ & 2 \leq c \end{aligned}$$

Therefore, we can choose any $c \geq 2$ with any $n_0 \geq 1$, for example $c = 2$, and $n_0 = 1$, and the definition of big-O will hold. QED

Notice all the backward pointing arrows!!! That's crucial for this being correct. Each step implies the step *above* it. Formally, this proof is really just an explanation of how you'd generate the proof (by writing those steps down in the opposite order). On the positive side, a nice thing about this proof is that it gives intuition of how you came up with the c and n_0.

Now, in contrast, here is a **wrong** "proof" written in the backward style:
Wrong *Proof that* $n + \cos(\pi n) \in O(n)$*:*

$$\begin{array}{rcll} n + \cos(\pi n) & \leq & cn & \\ 1 + \dfrac{\cos(\pi n)}{n} & \leq & c & \text{Dividing both sides by } n, \text{ since } n > 0. \\ \text{Therefore } 1 + 1 & \leq & c & \text{Because } \cos(x) \leq 1 \\ 2 & \leq & c & \text{Let } c = 2 \\ 2 & \leq & 2 & \text{True} \end{array}$$

What's wrong here is that the "proof" confuses what you know and what you are trying to prove. It ends up purporting to prove the $c \geq 2$, which isn't actually required, and it assumes that the big-O relationship holds, which is what you're trying to prove.

To see what can go wrong with this bogus, wrong style of writing "proofs", here's a "proof" that $1 + 1 = 3$:

$$\begin{array}{rcll} 1 + 1 & = & 3 & \\ x + x & = & 3x & \text{multiplying both sides by } x \\ e^x e^x & = & e^{3x} & \text{exponentiating both sides} \\ e^x & = & \dfrac{e^{3x}}{e^x} & \text{dividing both sides by } e^x \\ e^x \left(\dfrac{e^{3x}}{e^x} \right) & = & e^{3x} & \text{substituting for } e^x \text{ in the line } e^x e^x = e^{3x} \\ \ln \left(e^x \left(\dfrac{e^{3x}}{e^x} \right) \right) & = & \ln(e^{3x}) & \text{logarithm of both sides} \\ \ln(e^x) + \ln \left(\dfrac{e^{3x}}{e^x} \right) & = & 3x & \text{log identities} \\ x + \ln(e^{3x}) - \ln(e^x) & = & 3x & \text{log identities} \\ x + 3x - x & = & 3x & \text{log identities} \\ 3x & = & 3x & \text{True.} \end{array}$$

QED?!?

Notice how every step of this "proof" is mathematically valid. Each line derives logically from the lines above it. However, because this backward-style "proof" doesn't keep track of the implications needing to go backwards, it's easy to accidentally assume what you're trying to prove.

4 Not Big-Θ Proof [10 marks]

Prove that $n + n\cos(\pi n) \notin \Theta(n)$. You must give a formal proof, based on the formal definition of big-Θ. (If you're rusty with your trigonometry functions, don't panic. All you need to recall is that $\cos(0) = \cos(2\pi) = \cos(4\pi) = \cos(6\pi) = \ldots = 1$, and that $\cos(\pi) = \cos(3\pi) = \cos(5\pi) = \cos(7\pi) = \ldots = -1$.)

Solution : This question was marked less generously than the preceding proof (but both were marked consistently across the entire class, ensuring fairness). Fortunately, since we are *disproving* something, a good way

to approach this problem is via indirect proof (aka proof by contradiction), wherein you assume the opposite and see that it leads to a contradiction. This meant that all of you who instinctively wanted to start by assuming the definition of big-Θ started off on the right foot here (as opposed to Question 3, where you need to be more careful).

Overall, the results were bimodal, with about half the class doing very well (full marks, or nearly so), but the other half of the class doing not so well. A common error was trying to choose specific values of c and n_0 and showing that those don't work — that's NOT a valid way to disprove a big-O/Ω. It's like if you want to *dis*prove the statement "There is a good restaurant in Vancouver," you can't just go to a crappy restaurant and say, "See! This restaurant sucks!" You have to show somehow that *every* restaurant in Vancouver is bad. Similarly, to disprove that there exists c and n_0, etc., you must show that every choice for c and n_0 won't work.

If you had trouble with this problem, work to understand what you did wrong, and/or seek out help from Piazza/TAs/profs.

OK, here's a sample proof:

Proof that $n + n\cos(\pi n) \notin \Theta(n)$.

The proof is by contradiction.

Assume that $n + n\cos(\pi n) \in \Theta(n)$.

That implies that $n + n\cos(\pi n) \in \Omega(n)$.

By the definition of big-Ω (for variety, I'll use Lars's definition of big-Ω here instead of mine), there must exist $d > 0$ and n_0, such that $dn \leq n + n\cos(\pi n)$ for all $n \geq n_0$.

However, note that $n + n\cos(\pi n) = 0$ whenever n is odd. Therefore, no matter what values of d and n_0 are chosen, consider n_1 being a positive, odd integer greater than n_0. Then since $n_1 > n_0$, the following inequality must hold:

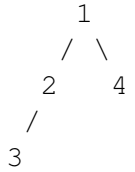
$$\begin{array}{rcll} dn_1 & \leq & n_1 + n_1\cos(\pi n_1) & \text{by the definition of big-}\Omega\text{, since } n_1 > n_0 \\ \text{Therefore, } dn_1 & \leq & 0 \quad \text{since } n_1 \text{ is odd} & \\ & & \text{which is a contradiction, since } d > 0 \text{ and } n_1 > 0. & \end{array}$$

Therefore, $n + n\cos(\pi n) \notin \Theta(n)$. QED

5 Second Best Heaps [10 marks]

A convenient feature of binary (min-)heaps is that the smallest element is always located at `heap[0]` (i.e., the root of the tree). Therefore, it's tempting to think that the **second** smallest element must be at location `heap[1]` or `heap[2]`, i.e., the two children of the root. Draw a counter-example to this claim. In other words, draw a binary min-heap here, that obeys all the properties required of a heap, and with all priority values being distinct, where the second smallest value is **not** located at one of the two children of the root. To make marking easier, please draw your heap in tree form, not as the array that would actually be used to store it.

Solution : Apologies again for the mistake on this problem. This was entirely my (Alan's) fault — just a pure mental lapse. the intention was to ask about the **third** smallest element, expecting a solution like:



However, with the problem being wrong, we gave everyone full credit (5 points out of 5) on this portion of the problem. A few people with particularly insightful comments (e.g., a proof that the problem was incorrect) got a bonus point or two. **(End of Solution)**

Given a min-heap `h` implementing a priority queue, fill in code to implement the `delete2ndMin` function below. The priority queue provides the usual methods: `void insert(Object o, int priority)`, which adds object `o` to the priority queue; and `Object deleteMin(int &priority)`, which returns the object with the lowest priority value, removes it from the priority queue, and also sets reference parameter `priority` to return the priority value the object had.

Your `delete2ndMin` function should act like `deleteMin`, except that it returns (and removes from the priority queue), the object with the 2nd smallest priority value. Your implementation must run in $O(\log n)$ time. **Note: Your implementation must not use any `if`, `switch`, or loop statements!** (Hint: Use the priority queue methods instead! A correct solution will be just a few lines of code.) You may assume there are no duplicate priority values, if that makes it easier for you to think about this problem. You may also assume that there are always at least two objects in the priority queue (so you don't have to check for empty queues).

```
Object delete2ndMin(Heap h) {
  // Your code goes here...

}
```

Solution : This is straightforward once you realize that you can use any constant number of heap operations, since they all run in $O(\log n)$ time. E.g.,

```
Object delete2ndMin(Heap h) {
  int temp1, temp2;
  Object smallest = h.deleteMin(temp1);
  Object second = h.deleteMin(temp2);
  h.insert(smallest, temp1); // Put smallest priority object back in.
  return second;
}
```

6 Pancake Stacks [10 marks]

In this problem, you will implement an ADT called a *pancake stack*. A pancake stack provides the normal stack operations (`push`, `pop`, `isEmpty`, etc.), but also provides a new operation `void flip(int n)`, which takes the top n items on the stack and "flips them over", i.e., reverses their order. For example, if you called `flip(3)` on the stack $\begin{matrix}1\\2\\3\\4\end{matrix}$ the result would be this stack: $\begin{matrix}3\\2\\1\\4\end{matrix}$, because the order of the top 3 elements on the stack has been reversed. If n is larger than the number of items in the stack, the entire stack gets reversed. (The intuition for the name "pancake stack" is that with a stack of pancakes, you can insert a spatula at any point in the stack and use it to flip over all pancakes above that point.)

You may assume a class `Stack` that provides a normal, non-pancake stack with these methods:

```
  void push(int); // Pushes an int onto the top of the stack
  int pop();  // Pops the top int from the stack and returns it.
  bool isEmpty();  // Returns true iff stack is empty.
```

For this problem, your class can have only one `Stack` variable as a member variable, and no other member variables. However, you are allowed to declare temporary `int` and `Stack` variables locally inside your methods if you need them.

Here's most of the code for the `PancakeStack` class, including implementations for the main methods except `flip`. Your task is to write the `flip` method.

```
  class PancakeStack {
    Stack s; // Storage for the pancake stack.
    // Note that you are NOT allowed to declare additional member variables.
  public:
    PancakeStack();
    ~PancakeStack();
    void push(int);
    int pop();
    int isEmpty();
    void flip(int); // Reverse order of the top number of elements in the stack
  }
  ...
  void PancakeStack::push(int n) { s.push(n); }

  int PancakeStack::pop() { return s.pop(); }

  int PancakeStack::isEmpty() { return s.isEmpty(); }

  void PancakeStack::flip() {
    // YOUR CODE GOES HERE...
```

Solution : The marking scheme for this question turned out rather harsh, but as with all problems, it was uniformly applied across the entire class, to ensure fairness. Correct solutions, of which there were quite a few, received full credit of course, but partial credit wasn't given out very generously. In particular, answers that failed to flip at all typically received 0 points, and loops that could underflow a stack (potentially causing crashes or undefined behaviour) lost points each time they occurred. (For example, very commonly, people would just loop up to n, regardless of what was in each stack.) On the generous side, although we specifically allowed only `int` and `Stack` variables, a few people attempted solutions using arrays of `int`s. Because the problem statement wasn't completely bullet-proof on this point, we still gave partial credit for such a solution, if it was correct. Anyway, here's a sample solution:

```
void PancakeStack::flip(int n) {
  Stack temp1, temp2;
  for (int i=0; i<n && !s.isEmpty(); i++) {
    // Pop off n elements, or until empty, and transfer to temp1.
    temp1.push(s.pop());
  }
```

```
  while (!temp1.isEmpty()) {
    // Reverse the order of the elements popped off.
    temp2.push(temp1.pop());
  }
  while (!temp2.isEmpty()) {
    s.push(temp2.pop());
  }
}
```

7 kth Smallest — Loop Invariant [10 marks]

The following code computes the kth smallest element within the specified part of an array — namely, from the elements `x[lo]` through `x[hi]` inclusive. (We will often use the notation `x[lo..hi]` to indicate the portion of the array from `x[lo]` to `x[hi]`.)

For example, if `kth_smallest` is called on the following array, with `lo=1`, `hi=4`, and `k=3`:

0	1	2	3	4	5
3	1	4	1	5	9

it will return the value 4, because the array elements in `x[1..4]` are 1, 4, 1, and 5, and the third smallest of these values is 4.

```
int kth_smallest(int x[], int lo, int hi, int k) {
// Computes the kth smallest element in the array x between
// positions lo and hi (inclusive).  In other words, the code
// looks at (and rearranges) array elements x[lo], x[lo+1], ...,
// x[hi], and finds the kth smallest element among them.
// You may assume without proof that 1 <= k <= hi-low+1
// In other words, there will always be at least k elements
// in the array slice x[lo..hi].

  if (lo == hi) {
    // Only one element in the range, so return it.  k must be 1
    return x[lo];
  }
  int i, p;
  p = lo;
  for (i=lo+1; i<=hi; i++) {
    // Loop Invariant:  x[lo+1 .. p] contains all elements of
    //                  x[lo+1 .. i-1] that are less than x[lo].
    if (x[i] < x[lo]) {
      p++;
      swap(x[p], x[i]);  // Swaps the contents of x[p] and x[i]
    }
  }
  swap(x[lo], x[p]);
  if (p-lo+1 == k) {
    // Everything in x[lo..p-1] is less than x[p],
    // so x[p] is the kth largest element in the array.
```

```
    return x[p];
  }
  if (p-lo+1 > k) {
    // kth smallest is in first part of the array.
    return kth_smallest(x, lo, p-1, k);
  } else {
    // kth smallest in the second part of the array.
    return kth_smallest(x, p+1, hi, k - (p-lo+1));
  }
}
```

For this problem, your task is to prove correct the loop invariant we've given you.

First, prove the base case here. (Remember, this is for the loop invariant, only! Don't worry about the recursion yet...)

Solution : Unfortunately, it looked as if a lot of students didn't even get to the last few problems. (Work on being able to do these problems faster — as you can see from the solutions, the problems had short and straightforward answers.) These last several problems weren't that hard if you understood the material being tested.

For loop invariants, the critical point is that the induction is on the number of times the loop body executes, **not** something about the size of the input, or some integer n or anything like that. A very common mistake here was trying to write something like, "The base case is when `lo==hi`..." WRONG! That has NOTHING to do with the loop invariant proof. (It is, however, one of the base cases for the next question, which is the overall proof of the function, so hopefully people re-used this idea there...)

So, a correct solution would start with something like "The base case is when the loop starts to execute for the first time." or "The base case is after zero iterations of the loop." (BTW, notice that these are generic sentences. They ALWAYS work as the base case for loop invariant proofs, because *that's how loop invariant proofs always are!*)

OK, here's the rest of a solution:
The base case is when the loop starts to execute for the first time. At that point `p == lo`, and `i == lo+1`. So, `x[lo+1 .. p]` and `x[lo+1 .. i-1]` are both just `x[lo+1 .. lo]`, which is an empty segment of the array. Therefore, `x[lo+1 .. p]` contains all elements of `x[lo+1 .. i-1]` that are less than `x[lo]`.

(End of Base Case)

Now, prove the inductive case for the loop invariant. You may assume that `swap` works correctly.

Solution : We assume the loop invariant holds at the top of the loop, i.e., that `x[lo+1 .. p]` contains all elements of `x[lo+1 .. i-1]` that are less than `x[lo]`.

The loop body has an `if` statement, so we have two cases to consider.

Case 1: If `x[i] < x[lo]`, then the body of the if statement executes. The first statement `p++;` increments `p`, so that now `x[lo+1 .. p-1]` contains all the elements of `x[lo+1 .. i-1]` that are less than `x[lo]`. Since `x[i] < x[lo]`, once we do the swap, know that `x[p] < x[lo]`, too. So now, `x[lo+1 .. p]` contains all elements of `x[lo+1 .. i]` that are less than `x[lo]`. After that, the loop increments `i`, which restores the loop invariant again, that `x[lo+1 .. p]` contains all elements of `x[lo+1 .. i-1]` that are less than `x[lo]`.

Case 2: If `x[i] >= x[lo]`, then the body of the if statement doesn't execute. Since we know that `x[i] >= x[lo]`, then we can extend the loop invariant property to include `x[i]`, as in: `x[lo+1 .. p]`

contains all elements of `x[lo+1 .. i]` that are less than `x[lo]`. After that, the loop increments `i`, which restores the loop invariant again, that `x[lo+1 .. p]` contains all elements of `x[lo+1 .. i-1]` that are less than `x[lo]`.

Since in either case, the loop invariant holds again at the bottom of the loop, that concludes the proof. QED

8 kth Smallest — Recursion Proof [10 marks]

This problem uses the same code as in Question 7, but you do not need to have solved that problem to do this one. For this problem, you will prove the overall function correct. Because the function is recursive, your proof must be an induction proof. You may assume the given loop invariant without proof, and that `swap` works correctly.

First, prove the base cases here. (Note that there are **two** very different base cases that your proof must handle, corresponding to the two situations where the function does **not** recurse.)

Solution : Base Case 1: When `lo==hi`, there is only one element in the range, and the problem states that we can assume without proof that `k=1`, so the 1st smallest element is the only element, which the code returns.

Base Case 2: The second base case occurs when `p-lo+1 == k` after the loop and swap. After the `for` loop, we know the loop invariant is true, that `x[lo+1 .. p]` contains all elements of `x[lo+1 .. i-1]` that are less than `x[lo]`. Furthermore, since the loop exited, we know that `i==hi+1`. Therefore, we know that `x[lo+1 .. p]` contains all elements of `x[lo+1 .. hi]` that are less than `x[lo]`. The code then swaps `x[lo]` and `x[p]`, so then `x[lo .. p-1]` contains all elements of `x[lo .. hi]` that are less than `x[p]`. In other words, `x[p]` is the `(p-lo+1)` th smallest element. Since `k == p-lo+1` in this case, the code correctly returns `x[p]`.

(End of Base Cases)

Now, prove the inductive cases for the recursive calls.

Solution : In the inductive cases, the code reaches the same point as in Base Case 2 above, and as noted above, `x[lo .. p-1]` contains all elements of `x[lo .. hi]` that are less than `x[p]`. However, `p-lo+1 != k`.

The code has an `if` statement, so there are two cases to consider. In the first case, if `p-lo+1 > k`, the code executes the "then" branch of the if statement. There are `p-lo` array elements in `x[lo .. p-1]`, and these are all of the elements smaller than `x[p]`, so the kth smallest element has to be in `x[lo .. p-1]`. This is a smaller slice of the array than `x[lo .. hi]`, so by the inductive hypothesis, the recursive call `kth_smallest(x, lo, p-1, k)` correctly returns the kth smallest element in `x[lo .. p-1]`, which has to be the kth smallest element in `x[lo .. hi]`.

In the second case, if `p-lo+1 < k`, the code executes the "else" branch of the if statement. There are `p-lo` array elements in `x[lo .. p-1]`, and these are all of the elements smaller than `x[p]`, so the kth smallest element has to be in `x[p+1 .. hi]`. Furthermore, since there `p-lo` smaller elements in `x[lo .. p-1]`, plus `x[p]`, then the kth smallest element in `x[lo .. hi]` will be the $k-(p-lo+1)$th smallest element in `x[p+1 .. hi]`. This is a smaller slice of the array than `x[lo .. hi]`, so by the inductive hypothesis, the recursive call `kth_smallest(x, p+1, hi, k - (p-lo+1))` correctly returns the $k-(p-lo+1)$th smallest element in `x[p+1 .. hi]`, which has to be the kth smallest element in `x[lo .. hi]`.

Since the code works correctly in both cases, the code is correct. QED

9 kth Smallest — Worst Case Runtime [10 marks]

This problem uses the same code as in Question 7, but you do not need to have solved that problem to do this one. For this problem, give tight big-O and big-Ω bounds on the **worst-case** runtime of `kth_smallest`. The size of the input n you should use in your analysis is the number of elements in the specified range in the array, i.e., `hi-lo+1`. You may assume that `swap` takes $\Theta(1)$ time per call. You do not need to show your work, but showing work can help you get partial credit.

Give a tight big-O upper bound on the **worst-case** runtime of `kth_smallest`:

Give a tight big-Ω lower bound on the **worst-case** runtime of `kth_smallest`:

Solution : Part of this problem (and the next one) is to realize that you can get big-O/big-Ω upper- and lower-bounds on either best-case or worst-case runtimes (or any other kind of analysis). On these problems, we can actually derive a big-Θ runtime bound, so the tight big-O upper bound and the tight big-Ω lower bound will be the same.

For this problem, the bound is $\Theta(n^2)$.

To solve this problem, there are really two separate steps: figuring out what the worst-case is, and then analyzing the code for that situation.

The worst case turns out to be when the partition step is completely lopsided (`p` ends up being equal to `lo` or `hi`). That yields a recurrence of $T(n) = T(n-1) + n$, which unrolls into the summation $\sum_{i=1}^{n} i$, which gives the $\Theta(n^2)$ bound.

10 kth Smallest — Best Case Runtime [10 marks]

This problem uses the same code as in Question 7, but you do not need to have solved that problem to do this one. For this problem, give tight big-O and big-Ω bounds on the **best-case** runtime of `kth_smallest`. The size of the input n you should use in your analysis is the number of elements in the specified range in the array, i.e., `hi-lo+1`. You may assume that `swap` takes $\Theta(1)$ time per call. You do not need to show your work, but showing work can help you get partial credit.

Give a tight big-O upper bound on the **best-case** runtime of `kth_smallest`:

Give a tight big-Ω lower bound on the **best-case** runtime of `kth_smallest`:

Solution : Similarly to the previous problem, we first have to figure out the best case. Note that the best case is **not** about when `lo==hi`, as that limits the size of the array (to one element), which isn't an asymptotic best case. However, what *is* an asymptotic best case is the other base case, when `p-lo+1 == k`. So, in this case, we spend linear time in the `for` loop, and then we're done. So, the solution if $\Theta(n)$.