Prolegomena unit outline:

- Algorithms and correctness (last week Wednesday and Friday)
- Algorithms and efficiency (today and Friday)
- Abstract data types (next week Monday)
- Data Structures (next week Wednesday and Friday)

Today and Friday:

- Go over Ex 1.(6 & 7)
- The general meaning of efficiency
- The analyses of bounded linear search, binary search, and selection sort
- The precise meaning of big-oh, big-theta, and big-omega
- The costs of elemental algorithms
- The analysis of quick sort
1.6 Write a loop invariant to capture the relationships among sequence, smallest_so_far, smallest_pos, and i in the following algorithm to find the smallest element in a sequence.

def find_smallest(sequence):
    smallest_so_far = sequence[0]
    smallest_pos = 0
    i = 1
    while i < len(sequence):
        if sequence[i] < smallest_so_far:
            smallest_pos = i
            smallest_so_far = sequence[i]
        i += 1
    return smallest_pos
1.7 State and prove a loop invariant to show that the following loop clears the list `sequence`, that is, it sets all of its positions to `None`. Your loop invariant should explain and relate the variables `sequence` and `i`.

```python
i = 0
while i < len(sequence):
    sequence[i] = None
    i += 1
```
From the correctness proof of bounded_linear_search:

By Invariant 1.c [\(i\) is the number of iterations], after at most \(n\) iterations, \(i = n\) and the guard will fail.

From the correctness proof of binary_search (rewritten):

Let \(i\) be the number of iterations completed. Suppose \(i \geq \lg n\). Then \(2^i \geq n\) and \(\frac{n}{2^i} \leq 1\).

By Invariant 3.b, \([\text{high} - \text{low} \leq \frac{n}{2^i}]\), we have \(\text{high} - \text{low} \leq 1\) and the guard fails.
def bounded_linear_search(sequence, P):
    found = False
    i = 0
    while not found and i < len(sequence):
        found = P(sequence[i])
        i += 1
    if found:
        return i - 1
    else:
        return -1

\[
T_{bls}(n) = a_0 + a_1(n + 1) + a_2 n + a_3 + \max(a_4, a_5) \\
= b_0 + b_1 n
\]
```python
def binary_search(sequence, T0, item):
    low = 0
    high = len(sequence)
    while high - low > 1:
        mid = (low + high) / 2
        compar = T0(item, sequence[mid])
        if compar < 0:  # item comes before mid
            high = mid
        elif compar > 0:  # item comes after mid
            low = mid + 1
        else:  # item is at mid
            assert compar == 0
            low = mid
            high = mid + 1
    if low < high and T0(item, sequence[low]) == 0:
        return low
    return -1
```

\[
T_{bs}(n) = c_0 + c_1(\lg n + 1) + c_2 \lg n + c_3 + \max(c_4, c_5) = d_0 + d_1 \lg n
\]
```python
def selection_sort(sequence, T0):
    for i in range(len(sequence)):
        min_pos = i
        min = sequence[i]
        for j in range(i + 1, len(sequence)):
            if T0(sequence[j], min) < 0:
                min = sequence[j]
                min_pos = j
        sequence[min_pos] = sequence[i]
        sequence[i] = min

T_{sel}(n) = f_1 + f_2 n + f_3 n^2
```
\exists T : D \rightarrow \mathbb{N} \text{ relating input to running time on some platform. Interpret the codomain } \mathbb{N} \text{ as natural numbers in some unit time.}

\dashv \forall T_{\text{absolute}} : \mathbb{N} \rightarrow \mathbb{N} \text{ relating input size to running time on some platform. Interpret the domain } \mathbb{N} \text{ as the number of items in the list (or other structure, for other algorithms).}

\exists T_{\text{worst}} : \mathbb{N} \rightarrow \mathbb{N} \text{ relating input size to the maximum running time on some platform for all inputs of the given size.}

\exists T_{\text{best}} : \mathbb{N} \rightarrow \mathbb{N} \text{ relating input size to the minimum running time on some platform for all inputs of the given size.}

\exists T_{\text{expected}} : \mathbb{N} \rightarrow \mathbb{N} \text{ relating input size to the expected value of the running time on some platform over all inputs of the given size.}
What is big-oh notation?

Big-oh is a way to categorize functions:

\[ O(g) \text{ is the set of functions that can be bounded above by a scaled version of } g. \]

\[ f(n) = O(g(n)) \text{ (or, more properly } f \in O(g)) \text{ means } \]

\[ \exists \ c, n_0 \in \mathbb{N} \text{ such that } \forall \ n \in [n_0, \infty), f(n) \leq cg(n) \]
Objections to and misconceptions of big-oh notation take forms such as

- Big-oh notation specifies only an upper bound of running time, which might be widely imprecise.
- Big-oh notation measures only the worst case, when the best case or the typical case might be much better.
- Big-oh ignores constants, which can greatly affect running time in practice.
- Algorithms that have the same big-oh category can have widely different running times in practice.
- Big-oh considers only the size of the input, when in fact other attributes of the input can greatly affect running time.
Coming up:

Due Today:
Finish reading Section 1.2, if you haven’t already.
(Exercises 1.(6 & 7) and the quiz should have been done before class)

Due Fri, Jan 21:
Read Sections 1.(3 & 4)
Do Exercises 1.(27, 28, 42, 43)
Take quiz

Due Tues, Jan 25:
Read Section 2.1
Do Exercise 1.11
Take quiz