Prolegomena unit outline:

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

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 merge sort and 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.

```python
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`.

```
i = 0
while i < len(sequence):
    sequence[i] = None
    i += 1
```

From the correctness proof of `bounded_linear_search`:

> *By Invariant 1.c [$\imath$ is the number of iterations], after at most n iterations, $\mathtt{i} = n$ 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, [$\mathtt{high} - \mathtt{low} \leq \frac{n}{2^i}$], we have $\mathtt{high} - \mathtt{low} \leq 1$ and the guard fails.*

```
def bounded_linear_search(sequence, P):
  $a_0$ found = False
        i = 0
        while not found and i < len(sequence):   $a_1(n+1)$
     $a_2 n$ found = P(sequence[i])
             i += 1
        if found: $a_3$
       $a_4$ return i - 1
        else :
       $a_5$ return -1
```

$$\begin{aligned} T_{bls}(n) &= a_0 + a_1(n+1) + a_2 n + a_3 + \max(a_4, a_5) \\ &= b_0 + b_1 n \end{aligned}$$

```
def binary_search(sequence, TO, item):
c0   low = 0
     high = len(sequence)
     while high - low > 1:   c1(lg n + 1)
c2 lg n  mid = (low + high) / 2
         compar = TO(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 TO(item, sequence[low]) == 0:   c3
 c4      return low
     else :
 c5      return -1
```

$$\begin{aligned} T_{bs}(n) &= c_0 + c_1(\lg n + 1) + c_2 \lg n + c_3 + \max(c_4, c_5) \\ &= d_0 + d_1 \lg n \end{aligned}$$

```
def selection_sort(sequence, TO):
    for i in range(len(sequence)):
        min_pos = i
        min = sequence[i]
        for j in range(i + 1, len(sequence)):
            if TO(sequence[j], min) < 0 :
                min = sequence[j]
                min_pos = j
        sequence[min_pos] = sequence[i]
        sequence[i] = min
```

$e_0 + e_1 n$

$e_2 n$

$e_3 n + e_4 \sum_{i=0}^{n-1} (n - i - 1)$

$e_5 \sum_{i=0}^{n-1} (n - i - 1)$

$$T_{sel}(n) \quad = \quad f_1 + f_2 n + f_3 n^2$$

- $\exists\ T : D \to \mathbb{N}$ relating input to running time on some platform. Interpret the codomain $\mathbb{N}$ as natural numbers in some unit time.
- $\nexists\ T_{\text{absolute}} : \mathbb{N} \to \mathbb{N}$ relating input size to running time on some platform. Interpret the domain $\mathbb{N}$ as the number of items in the list (or other structure, for other algorithms).
- $\exists\ T_{\text{worst}} : \mathbb{N} \to \mathbb{N}$ relating input size to the maximum running time on some platform for all inputs of the given size.
- $\exists\ T_{\text{best}} : \mathbb{N} \to \mathbb{N}$ relating input size to the minimum running time on some platform for all inputs of the given size.
- $\exists\ T_{\text{expected}} : \mathbb{N} \to \mathbb{N}$ 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*:

> *$O(g)$ is the set of functions that can be bounded above by a scaled version of g.*

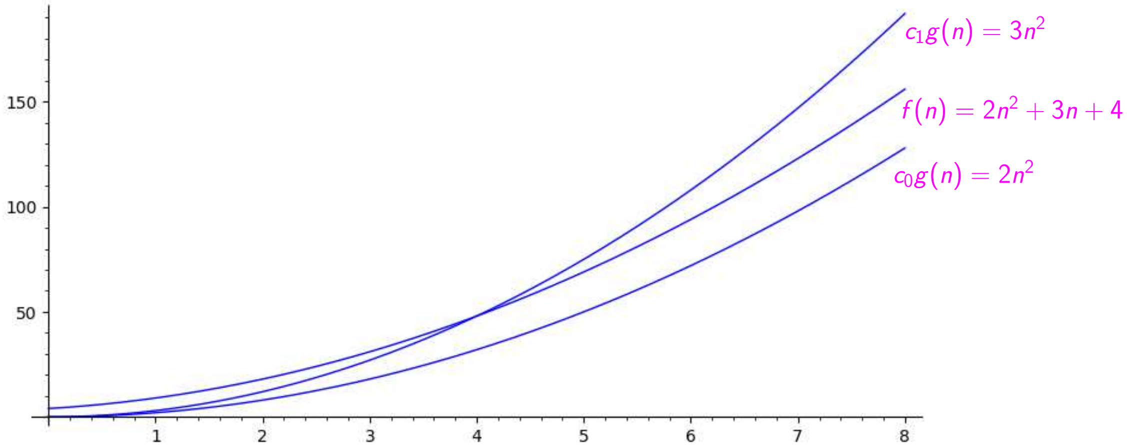
$f(n) = O(g(n))$ (or, more properly $f \in O(g)$) means

$$\exists\ c, n_0 \in \mathbb{N} \text{ such that } \forall\ n \in [n_0, \infty), f(n) \leq cg(n)$$

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.

$$\Theta(g) = \{f : \mathbb{N} \to \mathbb{N} \mid \exists\ c_0, c_1, n_0 \in \mathbb{N} \text{ such that } \forall\ n \in [n_0, \infty), c_0 g(n) \leq f(n) \leq c g(n)\}$$



Algorithmic element 1

Can you jump directly to the thing you're looking for?

Algorithmic element 2

Are you descending a binary tree of the data?

Algorithmic element 3

Do you need to touch every element in the data?

Algorithmic element 4

For every element, do you need to descend a tree, or for every element in the tree, do you touch every element?

Algorithmic element 5

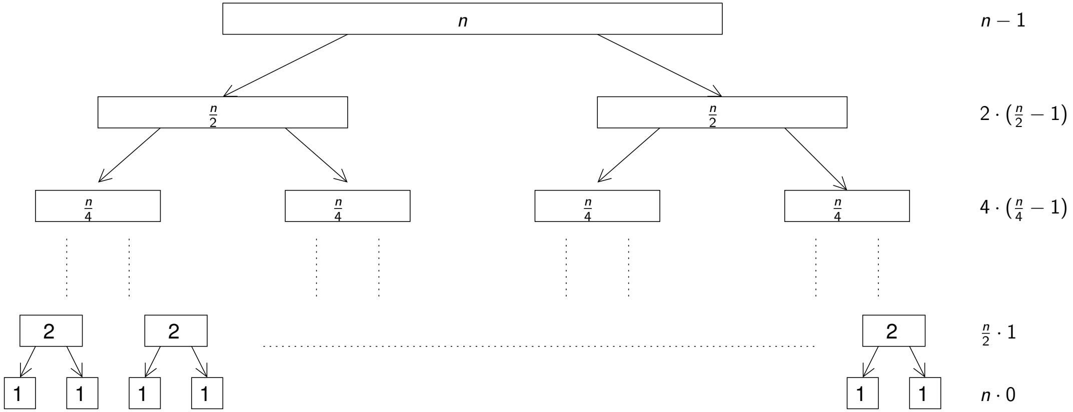
For every element in the data, do you need to a suboperation on the rest of the data?

Algorithmic element 6

Do you need to consider all combinations of input elements?

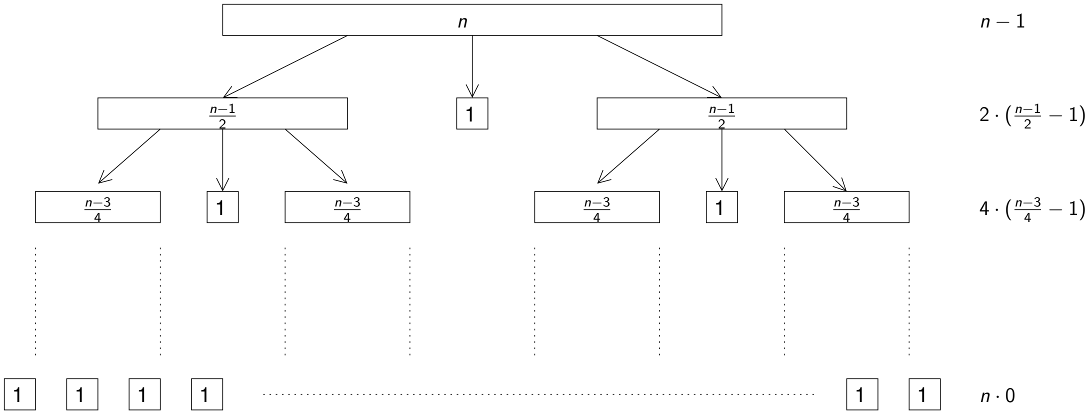
```
int merge_sort_r(int sequence[], int aux[], int low, int high)
{
  if (low + 1 >= high)
    return 0;
  else {
    int compars = 0;  // the number of comparisons
    int midpoint = (low + high) / 2; // index to the middle of the range
    int k, n;
    n = high - low;
    compars += merge_sort_r(sequence, aux, low, midpoint);
    compars += merge_sort_r(sequence, aux, midpoint, high);
    compars = merge(sequence, aux, low, high);
    return compars;
  }
}
```

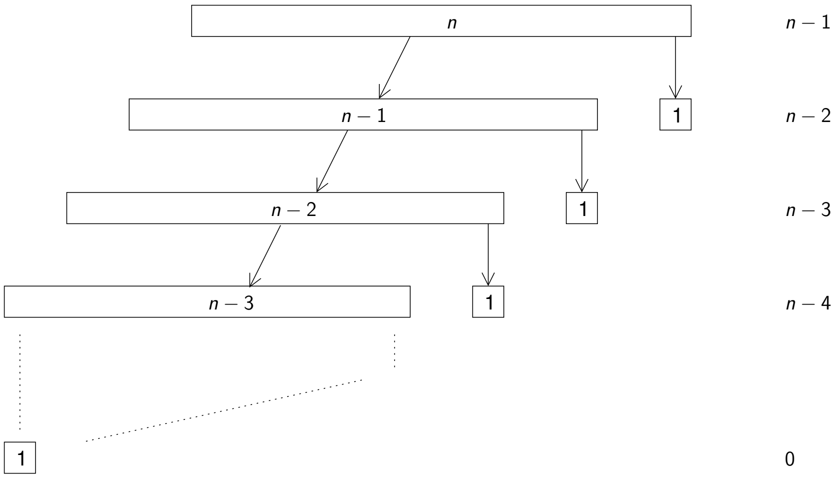
$$C_{ms}(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ n - 1 + 2C_{ms}(\frac{n}{2}) & \text{otherwise} \end{cases}$$



$$\begin{aligned} \textstyle\sum_{i=0}^{\lg n - 1} 2^i \cdot (\frac{n}{2^i} - 1) &= \textstyle\sum_{i=0}^{\lg n - 1} n - \sum_{i=0}^{\lg n - 1} 2^i \\ &= n \lg n - n + 1 \end{aligned}$$

```
int quick_sort_r(int sequence[], int low, int high)
{
  if (low + 1 >= high) return 0;
  int i, j, temp;
  int compars = 0;
  for (i = j = low; j < high-1; j++) {
    compars++;
    if (sequence[j] < sequence[high-1])
      {
        temp = sequence[j];
        sequence[j] = sequence[i];
        sequence[i] = temp;
        i++;
      }
  }
  temp = sequence[i];
  sequence[i] = sequence[j];
  sequence[j] = temp;
  return compars + quick_sort_r(sequence, low, i)
    + quick_sort_r(sequence, i+1, high);
}
```


n
$\frac{n-1}{2}$
1
$\frac{n-1}{2}$
$\frac{n-3}{4}$
1
$\frac{n-3}{4}$
$\frac{n-3}{4}$
1
$\frac{n-3}{4}$
1 1 1 1 1 1
$n-1$
$2 \cdot (\frac{n-1}{2} - 1)$
$4 \cdot (\frac{n-3}{4} - 1)$
$n \cdot 0$



$$(n-1)+(n-2)+(n-3)+\cdots+1+0=\sum_{i=1}^{n-1} i=\frac{n\cdot(n-1)}{2}=\frac{n^2-n}{2}$$

Coming up:

Due **Wednesday, Aug 31** *(end of day):*
Finish reading Section 1.2
(Exercises 1.(6 & 7) should have been done before class)
Take quiz, if you haven't already.

Due **Fri, Sept 2** *(end of day):*
Read Sections 1.(3 & 4)
Do Exercises 1.(17–18)
Take quiz

Due **Thurs, Sept 8** *(end of day):*
Read Section 2.1
Do Exercise 1.11
Take quiz