Sorting algorithms

- So what about cost of sorting?
- Assume complexity of sorting a list is O(sort(L))
- Then if we sort and search we want to know if sort(L) + log (len(L)) < len(L)
  - I.e. should we sort and search using binary, just use linear search
- Can't sort in less than linear time!

Amortizing costs

- But suppose we want to search a list k times?
- Then is sort(L) + k*log(len(L)) < k*len(L)?
  - Depends on k, but one expects that if sort can be done efficiently, then it is better to sort first
  - Amortizing cost of sorting over multiple searches may make this worthwhile
  - How efficiently can we sort?

Selection sort

```
def selSort(L):
    for i in range(len(L) - 1):
        minIndx = i
        minVal= L[i]
        j = i + 1
        while j < len(L):
            if minVal > L[j]:
                minIndx = j
                minVal= L[j]
            j += 1
        temp = L[i]
        L[i] = L[minIndx]
        L[minIndx] = temp
```

Analyzing selection sort

- Loop invariant
  - Given prefix of list L[0:i] and suffix L[i+1:len(L)-1], then prefix is sorted and no element in prefix is larger than smallest element in suffix

1. Base case: prefix empty, suffix whole list – invariant true
2. Induction step: move minimum element from suffix to end of prefix. Since invariant true before move, prefix sorted after append
3. When exit, prefix is entire list, suffix empty, so sorted

Analyzing selection sort

- Complexity of inner loop is O(len(L))
- Complexity of outer loop also O(len(L))
- So overall complexity is O(len(L)^2) or quadratic
- Expensive

```
def selSort(L):
    for i in range(len(L) - 1):
        minIndx = i
        minVal= L[i]
        j = i + 1
        while j < len(L):
            if minVal > L[j]:
                minIndx = j
                minVal= L[j]
            j += 1
        temp = L[i]
        L[i] = L[minIndx]
        L[minIndx] = temp
```