List Processing

list: a set of values of the same type
- arrays are perfect for this

basic operations
- insert item in list
- delete item from list
- search the list
- sort the list

insertion
- just insert at the end of the list
- complexity: $O(1)$

deletion
- search for the item to delete
- shift the remaining elements to the left
- complexity: $O(n)$ – on average, we'll compare $\frac{n}{2}$ items and shift $\frac{n}{2}$ elements

searching
- need 3 things
  - the list (array)
  - its length
  - the item to search
- writing a search function
  - return index of item if found, -1 if not found
    - pretty typical
- sequential/linear search
  - start at beginning of array
  - search until item is found or we reach end of array
- suppose our list has 1000 elements
  - if the item to search is near the front, search is fast
  - not so fast for an item that is near the end of the list
  - complexity: $O(n)$ - on average, we'll compare $\frac{n}{2}$ items
  - can we improve this? sure, sort
- ***HANDOUT*** searching_handout

sorting
- basic steps
  - comparisons
  - swaps
- ***HANDOUT*** sorts_handout
- bubble sort
  - sort list in increasing order
  - make successive swaps to move the largest element to the end of the array
    - the larger value "bubbles" to the end
  - inner loop controls number of comparisons per pass
  - outer loop controls number of passes through the array
  - to sort n elements, takes $n-1$ passes
  - can we optimize this?

how about if the array is already sorted?
maybe we can abort early (if no swaps are performed)

on average for a list of size n: $\frac{n(n-1)}{2}$ comparisons and $\frac{n(n-1)}{4}$ assignments

n=1000 → 500,000 key comparisons and 250,000 item assignments

complexity: $O(n^2)$

useful for small amounts of data

pass	comparisons						indices compared
		8	4	1	3	2	
1	4	4	1	3	2	8	0/1, 1/2, 2/3, 3/4
2	3	1	3	2	4	8	0/1, 1/2, 2/3
3	2	1	2	3	4	8	0/1, 1/2
4	1	1	2	3	4	8	0/1

i = outer loop = controls passes
j = inner loop = controls comparisons
j is used as an indexer (compare j to j+1 or j to j-1)

```
for (i=1; i<n; i++)
{
    for (j=1; j<=n-i; j++)
    {
       if (list[j] < list[j-1])
           swap(list[j], list[j-1])
    }
}
```

i	j	comp
1	1..4	[1] < [0], ...
2	1..3	...
3	1..2	...
4	1..1	...

```
for (i=1; i<n; i++)
{
    for (j=0; j<n-i; j++)
    {
       if (list[j] > list[j+1])
           swap(list[j], list[j+1])
    }
}
```

i	j	comp
1	0..3	[0] > [1], ...
2	0..2	...
3	0..1	...
4	0..0	...

selection sort

select the smallest element in the list and place it at the first position (a single swap)
starting at the second position, find the next-smallest and place it at the second position
so we are placing each item in its proper position in the list (starting at the front)
a sorted left side and unsorted right side is maintained
use same list for sorted/unsorted
so in unsorted side

find the location of the "smallest" element
move it to beginning of unsorted part of the list

on average for a list of size n: $\frac{n(n-1)}{2}$ comparisons and $3(n-1)$ assignments

n=1000 → 500,000 key comparisons and 3000 item assignments

hey, we've reduced the number of swaps considerably!
complexity: still $O(n^2)$
useful with small amounts of data, but when swapping is time-consuming

pass	comparisons					
-----	---------------					
		8	4	1	3	2
1	4	1	4	8	3	2
2	3	1	2	8	3	4
3	2	1	2	3	8	4
4	1	1	2	3	4	8

```
for (i=0; i<n-1; i++)
{
     minIndex = i;
     for (j=i+1; j<n; j++)
          if (list[j] < list[minIndex])
               minIndex = j;
     swap(list[i], list[minIndex]);
}
```

i	j	swap
--	--	------
0	1..4	0/1..4
1	2..4	1/2..4
2	3..4	2/3..4
3	4..4	3/4..4

insertion sort

use same list for sorted/unsorted
first part is sorted, second part is unsorted
place the first item in the unsorted side in its place in the sorted side
shift previous elements in sorted side forward until an appropriate slot is found
place item in its appropriate slot
on average for a list of size *n:* $\frac{n^2+3\text{n}-4}{4}$ comparisons and $\frac{n(n-1)}{4}$ assignments

n=1000 $\rightarrow$ 250,000 key comparisons and 250,000 item assignments

complexity: $O(n^2)$ (although almost $O(n)$ on almost sorted data!)
best when list is almost sorted

pass					

	8	4	1	3	2
1	4	8	1	3	2
2	1	4	8	3	2
3	1	3	4	8	2
4	1	2	3	4	8

you should know that for each sort, to sort *n* elements it takes $n-1$ passes
you should know that for each sort, a sorted and unsorted side is maintained

bubble: unsorted=left, sorted=right
select: unsorted=right, sorted=left

- insertion: unsorted=right, sorted=left
- you should know that a single element is trivially sorted
- you should also know:
    - bubble sort
        - on each pass, largest item in unsorted side “bubbles” to end of unsorted side
        - many swaps, many comparisons
    - select sort
        - on each pass, smallest item in unsorted side moves to beginning of unsorted side
        - few swaps, many comparisons
    - insertion sort
        - on each pass, first item in unsorted side moves to its proper place in sorted side
        - some swaps, some comparisons
- you are not expected to memorize any sort code
    - it will be provided for you on tests

sequential search on ordered list

- can abort search early if current element is greater than the one we wish to find
- again, on average, for a list of size *n*, takes $\frac{n}{2}$ key comparisons
    - but typically a bit better than sequential search on unordered list
- downside: insertion is more costly
- complexity: $O(n)$

binary search

- the list must be sorted
- uses a divide and conquer method
- usually applied to array based lists since the technique is to find the middle of the list
- middle element = $\frac{\text{first}+\text{last}}{2}$
    - e.g. 100 elements = (0+99)/2 = 99/2 = 49 (remember integer division!)
- if middle element is the item we're searching for, we're done
- otherwise, we split the list into two parts and select the appropriate part
    - first half if our item is less than the middle element
    - second half if our item is greater than the middle element
- we keep dividing until we find the element (or if it's not there at all)
- complexity: $O(lg(n))$