Searching & Sorting

1 Sorting

- Why sort?
- What are some common sorting algorithms?

1.1 Search

The answer to "Why sort?"
(also organization, but *why organize?*)

Common Search Techniques

- Linear Search (aka "Brute Force" Search)
- Binary Search
- Hash table/Dictionaries

1.2 Common Sorting Techniques

- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Merge Sort

Bubble sort Iterate through the array examining adjacent pairs of elements. If necessary, swap them to put them in the desired order (Brick Sort).

Insertion sort Iterate through the array placing the ith element with respect to the $i-1$ previous elements.

Selection sort Iterate through the array putting the ith smallest element in the ith location.

Merge sort Useful for sorting very large amounts of data. The basic algorithm:

1. Split the file into smaller files.
2. Sort the smaller files
3. Merge the sorted files

Quick sort Partition the list into smaller lists such that every element in the left partition is less than or equal to the right partition. Repeat Quicksort process on the partitions—frequently done using recursion.

1.3 Sorting Properties

- Comparisons
- Swaps
- Run time

1.4 Mergesort

How does it work?

Splits into pieces, then merges the pieces together in order.

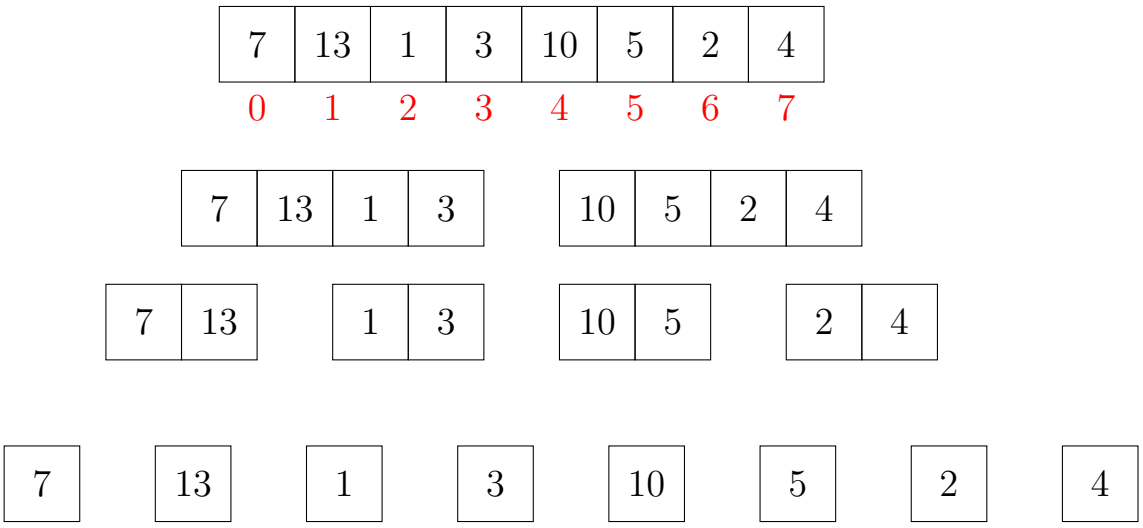
Originally performed on files – old computers didn't have much RAM (understatement).

Mergesort Diagram

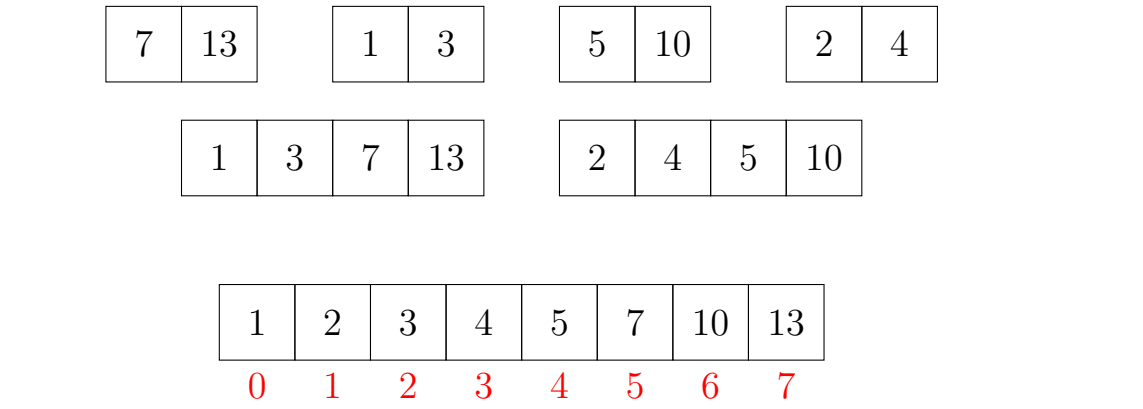
Initial array

7	13	1	3	10	5	2	4
0	1	2	3	4	5	6	7

Splitting



Merging



Q: How many subdivisions (levels)?

Q: How many subdivisions (levels)?
A: $2 \log_2 N$

Q: What is $\mathcal{O}(N)$?
A: $N \log_2 N$

Q: What is $\mathcal{O}(N)$ of Mergesort?

Q: What is $\mathcal{O}(N)$ of Mergesort?
A: $N \log_2 N$

Q: Why?
A: Look back at the diagram that shows behavior of Mergesort.

```
1  void Mergesort( aType a[], int first, int last )
2  {
3     int  middle;
4
5     cout << "\nMergesort:" << endl;
6     cout << "Mergesort::first: " << first << endl;
7     cout << "Mergesort::last:  " << last  << endl;
8
9     if( first < last )
10    {
11       middle = (first + last)/2;
12       cout << "Mergesort::middle:  " << middle << endl;
13       Mergesort( a, first, middle);
14       Mergesort( a, middle+1, last );
15       Merge( a, first, middle, middle+1, last );
16    }
17 }
```

```
1  void Merge( aType a[],
2              int firstLeft,  int lastLeft,
3              int firstRight, int lastRight )
4  {
5     aType tempArray[MAX_ARRAY];
6     int  index = firstLeft;
7     int  firstSave = firstLeft;
8
9     cout << "Merge::firstLeft:  " << firstLeft  << endl;
10    cout << "Merge::lastLeft:    " << lastLeft   << endl;
11    cout << "Merge::firstRight:  " << firstRight << endl;
12    cout << "Merge::lastRight:   " << lastRight  << endl;
```

```
1     while( (firstLeft <= lastLeft) &&
2            (firstRight <= lastRight)    ) {
3        if( a[firstLeft] < a[firstRight] )
4        {
5           tempArray[index] = a[firstLeft];
6           firstLeft++;
7        } else {
8           tempArray[index] = a[firstRight];
9           firstRight++;
10       }
11       index++;
12    }
13
14    while( firstLeft <= lastLeft ) {
15       tempArray[index++] = a[firstLeft];
16       firstLeft++;
17    }
18    while( firstRight <= lastRight ) {
19       tempArray[index++] = a[firstRight];
20       firstRight++;
21    }
22
23    for( index = firstSave ; index <= lastRight ; index++ )
24        a[index] = tempArray[index];
25 }
```

Output:

```
nA: 8
Initial array contents:
[ 7, 13, 1, 3, 10, 5, 2, 4 ]
Mergesort::first: 0 Mergesort::last:  7
Mergesort::middle:  3
Mergesort::first: 0 Mergesort::last:  3
Mergesort::middle:  1
Mergesort::first: 0 Mergesort::last:  1
Mergesort::middle:  0
Mergesort::first: 0 Mergesort::last:  0
Mergesort::first: 1 Mergesort::last:  1
[ 7, 13, 1, 3, 10, 5, 2, 4 ]
Mergesort::first: 2 Mergesort::last:  3
Mergesort::middle:  2
Mergesort::first: 2 Mergesort::last:  2
Mergesort::first: 3 Mergesort::last:  3
[ 7, 13, 1, 3, 10, 5, 2, 4 ]
[ 1, 3, 7, 13, 10, 5, 2, 4 ]
Mergesort::first: 4 Mergesort::last:  7
Mergesort::middle:  5
Mergesort::first: 4 Mergesort::last:  5
Mergesort::middle:  4
Mergesort::first: 4 Mergesort::last:  4
Mergesort::first: 5 Mergesort::last:  5
[ 1, 3, 7, 13, 5, 10, 2, 4 ]
Mergesort::first: 6 Mergesort::last:  7
Mergesort::middle:  6
Mergesort::first: 6 Mergesort::last:  6
Mergesort::first: 7 Mergesort::last:  7
[ 1, 3, 7, 13, 5, 10, 2, 4 ]
[ 1, 3, 7, 13, 2, 4, 5, 10 ]
[ 1, 2, 3, 4, 5, 7, 10, 13 ]

Final array contents:
[ 1, 2, 3, 4, 5, 7, 10, 13 ]
```

Output:

```
nA: 8
Initial array contents:
[ 7, 13, 1, 3, 10, 5, 2, 4 ]

Mergesort::first: 0 Mergesort::last:  7
Mergesort::middle:  3
Mergesort::first: 0 Mergesort::last:  3
Mergesort::middle:  1
Mergesort::first: 0 Mergesort::last:  1
Mergesort::middle:  0
Mergesort::first: 0 Mergesort::last:  0
Mergesort::first: 1 Mergesort::last:  1
Merge::firstLeft:  0    Merge::lastLeft:   0
Merge::firstRight: 1    Merge::lastRight:  1
Mergesort::first: 2 Mergesort::last:  3
Mergesort::middle:  2
Mergesort::first: 2 Mergesort::last:  2
Mergesort::first: 3 Mergesort::last:  3
Merge::firstLeft:  2    Merge::lastLeft:   2
Merge::firstRight: 3    Merge::lastRight:  3
Merge::firstLeft:  0    Merge::lastLeft:   1
Merge::firstRight: 2    Merge::lastRight:  3
Mergesort::first: 4 Mergesort::last:  7
Mergesort::middle:  5
Mergesort::first: 4 Mergesort::last:  5
Mergesort::middle:  4
Mergesort::first: 4 Mergesort::last:  4
Mergesort::first: 5 Mergesort::last:  5
Merge::firstLeft:  4    Merge::lastLeft:   4
```

```
Merge::firstRight: 5    Merge::lastRight:  5
Mergesort::first: 6 Mergesort::last:  7
Mergesort::middle:  6
Mergesort::first: 6 Mergesort::last:  6
Mergesort::first: 7 Mergesort::last:  7
Merge::firstLeft:  6    Merge::lastLeft:   6
Merge::firstRight: 7    Merge::lastRight:  7
Merge::firstLeft:  4    Merge::lastLeft:   5
Merge::firstRight: 6    Merge::lastRight:  7
Merge::firstLeft:  0    Merge::lastLeft:   3
Merge::firstRight: 4    Merge::lastRight:  7

Final array contents:
[ 1, 2, 3, 4, 5, 7, 10, 13 ]
```

1.5 Quicksort

- How does it work?
- How to choose the pivot value?

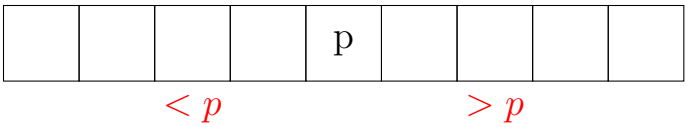
How does it work?

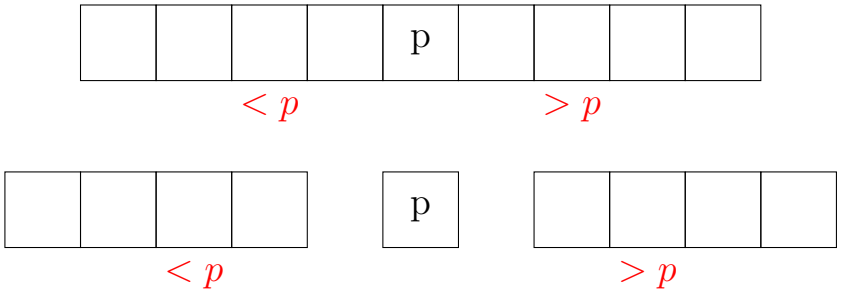
Chooses a *pivot value*, moves values less than the pivot value to the left and values greater than the pivot value to the right. Applies quicksort to the two "halves" recursively until done.

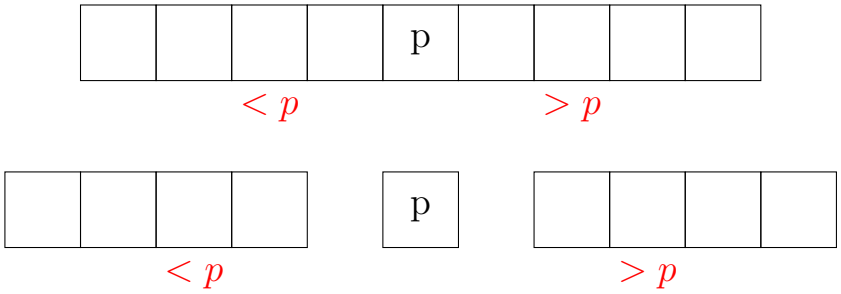
How to choose the pivot value?

Many ways to choose the pivot value. Using the first value is probably the easiest.

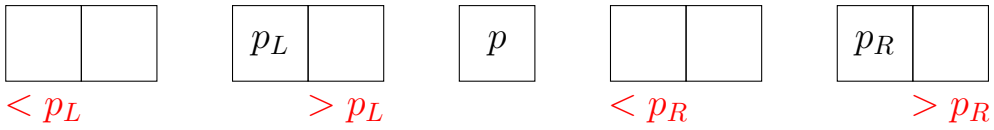
Array after subdividing about the pivot value, p.




p
$< p$
$> p$
p
$< p$
$> p$



Next subdivision (left subarray has a pivot, p_L and right subarray has a pivot, p_R)



Q: How many subdivisions (levels)?

Q: How many subdivisions (levels)?
A: $\log_2 N$

Q: What is $\mathcal{O}(N)$?
A: $N \log_2 N$

Q: What is $\mathcal{O}(N)$ of Quicksort?

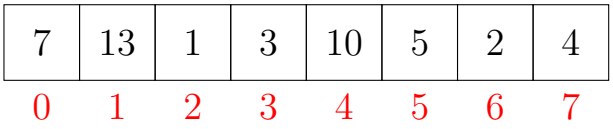
Q: What is $\mathcal{O}(N)$ of Quicksort?
A: $N \log_2 N$

Q: Why?
A: Look back at diagram that shows behavior of Quicksort.

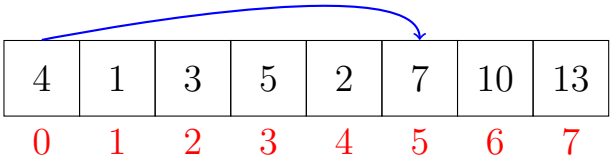
```
1  void Quicksort( aType a[], int first, int last )
2  {
3     int  pivot;  // index of pivot value
4
5     cout << "\nQuicksort:" << endl;
6     cout << "Quicksort::first: " << first << endl;
7     cout << "Quicksort::last:  " << last  << endl;
8
9     if( first < last )
10    {
11       pivot = Pivot( a, first, last );
12       cout << "Quicksort::pivot:  " << pivot << endl;
13       Quicksort( a, first, pivot-1 );
14       Quicksort( a, pivot+1, last );
15    }
16 }
```

```
1  int Pivot( aType a[], int first, int last )
2  {
3     int  p = first;           // pivot index
4     int  pivot = a[first];    // pivot value
5
6     cout << "\nPivot:" << endl;
7     cout << "Pivot::first: " << first << endl;
8     cout << "Pivot::last:  " << last  << endl;
9
10    for( int i = first+1 ; i <= last ; i++ ) {
11       if( a[i] <= pivot ) {
12          p++;
13          Swap( a[i], a[p] );
14       }
15    }
16
17    Swap( a[p], a[first] );
18
19    return p;
20 }
```

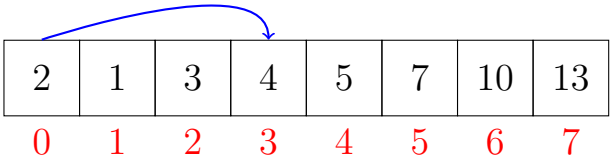
Initial array



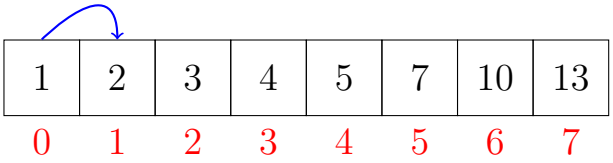
Pivot value: 7



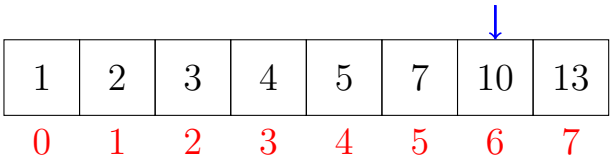
Pivot value: 4



Pivot value: 2



Pivot value: 10



Output (old format):

```
nA: 8
Random data:
Initial array contents:
[ 7, 13, 1, 3, 10, 5, 2, 4 ]
Quicksort::first: 0 Quicksort::last:  7
Pivot::first: 0 Pivot::last:  7
Quicksort::pivot:  5
Quicksort::first: 0 Quicksort::last:  4
Pivot::first: 0 Pivot::last:  4
Quicksort::pivot:  3
Quicksort::first: 0 Quicksort::last:  2
Pivot::first: 0 Pivot::last:  2
Quicksort::pivot:  1
Quicksort::first: 0 Quicksort::last:  0
Quicksort::first: 2 Quicksort::last:  2
Quicksort::first: 4 Quicksort::last:  4
Quicksort::first: 6 Quicksort::last:  7
Pivot::first: 6 Pivot::last:  7
Quicksort::pivot:  6
Quicksort::first: 6 Quicksort::last:  5
Quicksort::first: 7 Quicksort::last:  7
Final array contents:
[ 1, 2, 3, 4, 5, 7, 10, 13 ]
Total swaps:  13
```

Output:

```
nA: 8
Random data:
Initial array contents:
[ 7, 13, 1, 3, 10, 5, 2, 4 ]
Quicksort::first: 0 Quicksort::last:  7
Pivot::first: 0 Pivot::last:  7
Pivot::pivot: 7
Quicksort::pivot index:  5
Quicksort::A[]: [ 4, 1, 3, 5, 2, 7, 10, 13 ]

Quicksort::first: 0 Quicksort::last:  4
Pivot::first: 0 Pivot::last:  4
Pivot::pivot: 4
Quicksort::pivot index:  3
Quicksort::A[]: [ 2, 1, 3, 4, 5, 7, 10, 13 ]

Quicksort::first: 0 Quicksort::last:  2
Pivot::first: 0 Pivot::last:  2
Pivot::pivot: 2
Quicksort::pivot index:  1
Quicksort::A[]: [ 1, 2, 3, 4, 5, 7, 10, 13 ]

Quicksort::first: 0 Quicksort::last:  0
Quicksort::first: 2 Quicksort::last:  2
Quicksort::first: 4 Quicksort::last:  4
Quicksort::first: 6 Quicksort::last:  7
Pivot::first: 6 Pivot::last:  7
Pivot::pivot: 10
Quicksort::pivot index:  6
```

```
Quicksort::A[]: [ 1, 2, 3, 4, 5, 7, 10, 13 ]

Quicksort::first: 6 Quicksort::last:  5
Quicksort::first: 7 Quicksort::last:  7
Final array contents:
[ 1, 2, 3, 4, 5, 7, 10, 13 ]
Total swaps:  13
```

```
Sorted data (low to high):
Initial array contents:
[ 1, 2, 3, 4, 5, 7, 10, 13 ]
Quicksort::first: 0 Quicksort::last:  7
Pivot::first: 0 Pivot::last:  7
Quicksort::pivot:  0
Quicksort::first: 0 Quicksort::last:  -1
Quicksort::first: 1 Quicksort::last:  7
Pivot::first: 1 Pivot::last:  7
Quicksort::pivot:  1
Quicksort::first: 1 Quicksort::last:  0
Quicksort::first: 2 Quicksort::last:  7
Pivot::first: 2 Pivot::last:  7
Quicksort::pivot:  2
Quicksort::first: 2 Quicksort::last:  1
Quicksort::first: 3 Quicksort::last:  7
Pivot::first: 3 Pivot::last:  7
Quicksort::pivot:  3
Quicksort::first: 3 Quicksort::last:  2
Quicksort::first: 4 Quicksort::last:  7
Pivot::first: 4 Pivot::last:  7
Quicksort::pivot:  4
Quicksort::first: 4 Quicksort::last:  3
Quicksort::first: 5 Quicksort::last:  7
Pivot::first: 5 Pivot::last:  7
Quicksort::pivot:  5
Quicksort::first: 5 Quicksort::last:  4
Quicksort::first: 6 Quicksort::last:  7
Pivot::first: 6 Pivot::last:  7
Quicksort::pivot:  6
Quicksort::first: 6 Quicksort::last:  5
Quicksort::first: 7 Quicksort::last:  7
Final array contents:
[ 1, 2, 3, 4, 5, 7, 10, 13 ]
Total swaps:  20
```

```
Sorted data (high to low):
Initial array contents:
[ 13, 10, 7, 5, 4, 3, 2, 1 ]
Quicksort::first: 0 Quicksort::last:  7
Pivot::first: 0 Pivot::last:  7
Quicksort::pivot:  7
Quicksort::first: 0 Quicksort::last:  6
Pivot::first: 0 Pivot::last:  6
Quicksort::pivot:  0
Quicksort::first: 0 Quicksort::last:  -1
Quicksort::first: 1 Quicksort::last:  6
Pivot::first: 1 Pivot::last:  6
Quicksort::pivot:  6
Quicksort::first: 1 Quicksort::last:  5
Pivot::first: 1 Pivot::last:  5
Quicksort::pivot:  1
Quicksort::first: 1 Quicksort::last:  0
Quicksort::first: 2 Quicksort::last:  5
Pivot::first: 2 Pivot::last:  5
Quicksort::pivot:  5
Quicksort::first: 2 Quicksort::last:  4
Pivot::first: 2 Pivot::last:  4
Quicksort::pivot:  2
Quicksort::first: 2 Quicksort::last:  1
Quicksort::first: 3 Quicksort::last:  4
Pivot::first: 3 Pivot::last:  4
Quicksort::pivot:  4
Quicksort::first: 3 Quicksort::last:  3
Quicksort::first: 5 Quicksort::last:  4
Quicksort::first: 6 Quicksort::last:  5
Quicksort::first: 7 Quicksort::last:  6
Quicksort::first: 8 Quicksort::last:  7
Final array contents:
[ 1, 2, 3, 4, 5, 7, 10, 13 ]
Total swaps:  43
```

1.6 Comparison of Mergesort and Quicksort

```
void Mergesort( aType a[], int first, int last )
{
   int  middle;

   if( first < last )
   {
      middle = (first + last)/2;
      Mergesort( a, first, middle);
      Mergesort( a, middle+1, last );

      Merge( a, first, middle, middle+1, last );
   }
}
```

```
void Quicksort( aType a[], int first, int last )
{
   int  pivot;  // index of pivot value

   if( first < last )
   {
      pivot = Pivot( a, first, last );

      Quicksort( a, first, pivot-1 );
      Quicksort( a, pivot+1, last );
   }
}
```

1.7 Visualization of Sorting Algorithms

Visualising Sorting Algorithms
https://visualgo.net/bn/sorting

Visual Sort Algorithms on Canvas
http://www.joshuakehn.com/blog/static/sort.html

1.8 Mergesort in OCaml

```
let rec merge x y =
  match x, y with
    [], l -> l
  | l, [] -> l
  | hx::tx, hy::ty ->
      if hx < hy
        then hx :: merge tx (hy :: ty)
        else hy :: merge (hx :: tx) ty

let rec msort l =
  match l with
    [] -> []
  | [x] -> [x]
  | _ ->
      let left = take (length l / 2) l
      and right = drop (length l / 2) l in
        merge (msort left) (msort right)
```

1.8.1 OCaml

"Caml" was originally an acronym for *Categorical Abstract Machine Language*.

The Objective Caml language and implementation, first released in 1996 and renamed to OCaml in 2011. Objective Caml was the first language to combine the full power of object-oriented programming with ML-style static typing and type inference.[1]

https://ocaml.org

[1] A History of OCaml

1.8.2 An explanation

```
let rec merge x y =     (* Two lists:  x and y *)
  match x, y with
    [], l -> l          (* empty, non-empty    *)
  | l, [] -> l          (* non-empty, empty    *)
  | hx::tx, hy::ty ->   (* head and tail       *)
      if hx < hy        (* smallest head value *)
        then hx :: merge tx (hy :: ty)
        else hy :: merge (hx :: tx) ty
                        (* join head with merge of rest *)

let rec msort l =
  match l with
    [] -> []            (* empty returns empty    *)
  | [x] -> [x]          (* one element            *)
  | _ ->                (* split, then merge      *)
      let left = take (length l / 2) l
      and right = drop (length l / 2) l in
        merge (msort left) (msort right)
```

1.9 Another look at Quicksort

1.9.1 Quicksort Algorithm

http://en.wikipedia.org/wiki/Quicksort

function quicksort(array)
  var list *less*, *greater*
  if length(array) ≤ 1
    return array // zero or one elements is already sorted
  select and remove a pivot value, *pivot*, from array
  for each x in array
    if $x \leq$ *pivot* then append x to *less*
    else append x to *greater*
  return concatenate(quicksort(*less*), *pivot*, quicksort(*greater*))

1.9.2 Quicksort as a Functional Program

```
listPtr Quicksort( listPtr l )
{
   if( l == NULL )
      return NULL;
   else
   {
      int pivot = head(l);
      listPtr left  = filter( lessThan, pivot, tail(l) );
      listPtr right = removeFrom( lessThan, pivot, tail(l) );

      return cat( Quicksort(left), cons(pivot, Quicksort(right)) );
   }
}
```

Is this a large departure from what we have done in the past?

Functions:

1. `head()`
2. `tail()`
3. `cons()`
4. `cat()`
5. `filter()`
6. `removeFrom()`