Sorting, Ranking, Indexing, Selecting

- The definitions of *sorting*, *ranking*, and *indexing* should be clear from the following example:

Original	Indexed	Ranked	Sorted
3	6	4	0
1	2	2	1
2	3	3	2
6	1	6	3
4	5	5	4
0	4	1	6

- If we can index and sort, we can rank: if v is the original array, v_i is the indexed array, v_r is the ranked array, and v_s is the sorted array, then indexing v_i gives v_r.

  In Matlab we can sort and rank an array `v` as follows:

```
[vs, vi] = sort(v);
[x, vr]  = sort(vi);
```

  Note that `vs = v(vi)` and `vs(vr) = v`. Also `x` contains the integers $1, 2, \ldots$ (up to the length of `x`).

- *Selection* is the problem of finding the m^{th} largest element in v. For instance, to find the median we perform selection with $m = \lfloor n/2 \rfloor$ (if n is odd and we index from zero).

  Selection from an array that has already been sorted is trivial, but there are efficient ways to do selection that do not require complete sorting.

Algorithmic concepts

- *Complexity:* The number of operations required to calculate a given quantity (e.g. a sorted list) typically depends on the amount of data being manipulated. For sorting, the amount of data is the number n of values in the list to be sorted. Complexity describes how the operation count grows with n.

- Complexity is measured in terms of the highest order term in n. For example, if it takes exactly $6n^2 + 3n + 5$ operations to calculate the result, the algorithm is said to be quadratic.

- In determining complexity, the definition of "operation" must be specified. In some cases arithmetic operations are dominant, while in other cases operations that move data around in the computer's memory are dominant.

- *In-place:* The data being manipulated by an algorithm occupy a certain amount of storage on the computer. If the algorithm requires negligible additional storage space, the calculation is said to be done "in-place".

Finding extreme values

- Finding extreme values (i.e. the maximum and the minimum) is a special case of selection.

  Extreme values can be located by cycling once through the array values. Thus the complexity is linear in the array length, and the dominant operation is the arithmetic comparison ($<$ or $>$). This is an in-place calculation.

  In C, the following functions find the maximum and the minimum. The related problems of finding the indices where the maximum and minimum occur are also addressed.

```
double max(double* v, const int n)
{
    double x = v[0];
    for (int i=1; i<n; ++i) x = (v[i] > x) ? v[i] : x;
    return x;
}

double min(double* v, const int n)
{
    double x = v[0];
    for (int i=1; i<n; ++i) x = (v[i] < x) ? v[i] : x;
    return x;
}

int maxloc(double* v, const int n)
{
  int k = 0;
  for (int i=1; i<n; ++i) k = (v[i] > v[k]) ? i : k;
  return k;
}

int minloc(double* v, const int n)
{
  int k = 0;
  for (int i=1; i<n; ++i) k = (v[i] < v[k]) ? i : k;
  return k;
}
```

In Octave these are built-in:

```
## a is the maximum value and b is the index of a in v.
[a,b] = max(v);

## a is the minimum value and b is the index of a in v.
[a,b] = min(v);
```

As an illustration, here is how you would implement these functions in Octave using loops if they were not built-in:

```
function [a,b] = Max(v)

  a = v(1);
  b = 1;
  for i=2:length(v)
    if (v(i) > a)
      a = v(i);
      b = i;
    endif
  endfor

endfunction


function [a,b] = Min(v)

  a = v(1);
  b = 1;
  for i=2:length(v)
    if (v(i) < a)
      a = v(i);
      b = i;
    endif
  endfor

endfunction
```

Straight Insertion

- Straight insertion is a simple but slow in-place sorting algorithm. It should only be used with very small lists (a rule of thumb is to use it if there are at most 8 values).

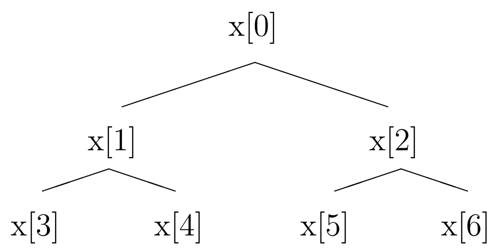
  The basic idea is to move from left to right, at each step locating the smallest value to the right of the current location, and then swap with the value in the current location.

3	2	1	8	4	5	7	Position 1 (swap 3 and 1).
1	2	3	8	4	5	7	Position 2 (nothing to swap).
1	2	3	8	4	5	7	Position 3 (nothing to swap).
1	2	3	8	4	5	7	Position 4 (swap 8 and 4).
1	2	3	4	8	5	7	Position 5 (swap 8 and 5).
1	2	3	4	5	8	7	Position 6 (swap 8 and 7).
1	2	3	4	5	7	8	Sorted array.

- It's easy to see that $n(n-1)/2$ floating point comparisons are the dominant operation, thus this has quadratic complexity. If you run the straight insertion algorithm on a list of length 4000, and then on a list of length 8000, you will see that the run time increases by around a factor of 4.

Heapsort

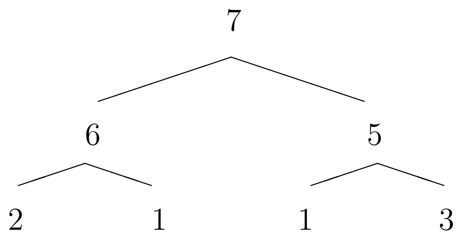
- The heapsort is an in-place sorting routine with complexity $n\log(n)$ – a great improvement on straight insertion. It is an example of a "tree-structured algorithms", and makes use of a data structure called a *heap*.
- A heap is a special case of the *binary tree*, an example of which is shown below:



- Notice how a binary tree can be stored and accessed efficiently as an array: the pair of nodes immediately below $x[i]$ ($x[i]$'s *children*) are nodes $x[2i+1]$ and $x[2i+2]$. The node above $x[i]$ ($x[i]$'s *parent*) is node $x[(i-1)/2]$ (where $(i-1)/2$ is the integer quotient, obtained by right shifting the bits, which is equivalent to the greatest integer less than the real-valued quotient).
- A *heap* is a binary tree in which each node is greater than or equal to all of the nodes that come below it (equivalently greater than or equal to its children). In that sense it is partially sorted.
- The first step of the heapsort algorithm is to convert the array that is to be sorted into a heap. This is done by cycling through the nodes from the bottom to the top. Each node is pushed down toward the bottom of the tree by exchanging it with nodes that are greater than it. This process continues until a point is reached where both of the children are smaller than the original node. The following shows the steps of converting the array $(3, 1, 5, 2, 6, 1, 7)$ into a heap:

```
3 1 5 2 6 1 7
3 1 7 2 6 1 5
3 6 7 2 1 1 5
7 6 5 2 1 1 3
```

The heap that we get is:



- Next we need to obtain a completely sorted array from the heap. This is done by repeatedly applying the "pop and sift" procedure. The largest element is at the top of the heap, so first we pull that element off the heap and store it somewhere temporarily. Next we pop the last element from the heap and put it at the top. Then we sift this element down, successively promoting the greater of its two children, until it reaches its proper level.

- Each iteration through the pop and sift process, the heap becomes one node smaller. This clears space at the end of the heap that we can re-use as storage for our sorted array. This means that at the end of each cycle, the element that we pulled off of the top of the heap at the beginning of the cycle can be stored in what was previously the last position in the heap.

  The following shows the state of the array after each cycle of the heap sorting part of the algorithm.

  ```
  7 6 5 2 1 1 3
  6 3 5 2 1 1 7
  5 3 1 2 1 6 7
  3 2 1 1 5 6 7
  2 1 1 3 5 6 7
  1 1 2 3 5 6 7
  ```

- *Complexity:* The tree formed from a list of length n has on the order of $\log_2(n)$ levels. Each of the n steps of building the original heap and each of the n iterations of the "pop and sift" procedure involves a comparison and possibly a swap for each level of the tree. Thus the complexity of the algorithm is $n\log(n)$.

Quicksort
The Quicksort algorithm is a recursive approach to sorting. The basic idea is to pick any element x from the list (x is called the *partition element*), then partition the list into the

elements V_1 that are smaller than x and the elements V_2 that are greater than x. Recursively sort V_1 and V_2, then concatenate the results, putting x in between.

Complexity depends on how well-balanced V_1 and V_2 are in size. If their sizes are roughly equal, then $\log_2 n$ splits are required. To carry out a split, the elements in V_1 and V_2 must be identified, which is a linear operation. So the overall complexity is $n \log_2 n$. This is the same as for heapsort, but it turns out to be a bit faster in practice.

The problem is that we can't guarantee that the split is balanced. In the worst case, suppose that the list is already sorted, and we always select the first element of the list as x. In this case one of the sublists is always empty, and it's easy to see that the complexity is quadratic.

By picking the partition element at random, it is very unlikely that the worst case will occur. Thus the average complexity is still better than heapsort. However the worst case complexity remains n^2.

An efficient implementation of quicksort depends on how the sublists V_1 and V_2 are identified. This is the best approach (assume that x has been removed from V):

1. Find the least index i_1 such $V_{i_1} > x$.
2. Find the greatest index i_2 such that $V_{i_2} < x$.
3. If $i_1 < i_2$, swap V_{i_1} and V_{i_2}, return to 1. If $i_1 \geq i_2$, stop.

When i_1 and i_2 (called *sentinels*) cross the following holds:

$$\begin{aligned} V_1, \ldots, V_{i_1-1} &\leq x \\ V_{i_2+1}, \ldots, V_n &\geq x \end{aligned}$$

Thus we can split the list by taking $V_1, \ldots, V_{i_2}$ as the first list and $V_{i_2+1}, \ldots, V_n$ as the second list.

3	1	8	9	5	0	2	7	Pop off the 3.
	1	8	9	5	0	2	7	Swap 8 and 2.
	1	2	9	5	0	8	7	Swap 9 and 0.
	1	2	0	5	9	8	7	The sentinels have crossed, $i_1 = 4$, $i_2 = 3$.
	1	2	0	5	9	8	7	Recur on (1, 2, 0) and (5, 9, 8, 7).
0	1	2	3	5	7	8	9	Put the 3 back in.

Indexing and Ranking

To construct an index table for an array, we apply any sorting algorithm, but instead of manipulating the list directly, we just keep track of the manipulations in an index vector.

Selection

- Selection is the problem of locating the m^{th} smallest element in a list. This is clearly important in statistics, since it gives the m^{th} order statistic, which is commonly used as to estimate the m/n quantile.

- One way to select the m^{th} smallest element is to sort the list from least to greatest, and then extract the m^{th} element. This may be the best thing to do if you want to select from the same list for many different values of m. If a single element is to be selected from a long list, sorting the entire list entails a lot of extra work.

- An efficient way to do selection is to borrow the idea of the Quicksort algorithm. The key is that at each level of the recursion, we only need to continue sorting one of the two subarrays, since we know which subarray contains the element that we are selecting. For example, we can locate the median (using the crude definition $Q(k/n) = X_{(k)}$) in the following array.

5	2	7	3	4	8	1	Pop off the 5.
	2	7	3	4	8	1	Swap 7 and 1.
	2	1	3	4	8	7	The sentinels have crossed.
2	1	3	4	5	8	7	Insert the 5.
2	1	3	4				Median is largest element of this list.