Sorting

- Basic sorting algorithms
  1) Selection
  2) Bubble sort
  3) **Insertion sort**
  4) **Distribution counting**
  5) **Quicksort**

- Advanced sorting algorithms
  1) Heapsort
  2) **Shell sort**
  3) **Radix sort**
  4) **Radix exchange sort**

Sorting - 1

Insertion sort - path to shell sort

- Make only one pass over the original list
  - As each element is encountered, insert it into its appropriate position in the output list, searching the output list from right to left.
  - This may require moving many items in the output list
- Insertion sort trades off repeated passes over the input list for many moves of data in the output list and comparisons of elements from the input list with the current output list. It is still an $O(n^2)$ sort.
- But number of move operations depends on the input, so insertion sort can be faster than other $O(n^2)$ sorts such as bubble sort or selection sort
- Insertion sort is also stable - it maintains order of tied elements from input list.

Sorting - 2

Insertion sort - example

```
6 5 8 2 3 1 7
- 5 8 2 3 1 7
- - 8 2 3 1 7
- - - 2 3 1 7
- - - - 3 1 7
- - - - - 1 7
- - - - - - 7
- - - - - - -
```

left-to-right	r-l
6 - 0 moves, 0 comparisons	1/0
5 6 - 1 move, 1 comp	1/1
5 6 8 - 1 move, 2 comps	1/1
2 5 6 8 - 3 moves, 1 comp	3/3
2 3 5 6 8 - 3 moves, 2 comps	3/4
1 2 3 5 6 8 - 5 moves, 1 comp	5/5
1 2 3 5 6 7 8 - 1 move, 6 comps	1/2

Notice: For i'th item from input list, we perform a total of i moves and comparisons with the output list.

Shell sort

- Insertion sort only works well if the table is originally "almost" sorted.
- For example, if no element is further than 4 positions from its final position, then the insertion of an element will require at most 4 comparisons and moves - making the algorithm linear.
- Can make insertion sort very efficient by wrapping it in a third outer loop!
- Result is called **shell sort**
- Insertion sort is inefficient because if an element has to move a long distance, it moves that distance one hop at a time.
- In shell sort, we will move an element many hops at a time

Shell sort

- Let the input list be A.
- Imagine partitioning the list into five nonoverlapping lists:

  A[1], A[6], A[11],...

  A[2], A[7], A[12], ...

  A[3], A[8], A[13], ...
- Sort each of these interlaced lists independently using insertion sort
- Hope is that small elements initially on the right will move in a small number of hops over to the left.

  A[j] <-> A[j+5]

Shell sort

- After the individual interlaced lists are sorted, we need to compare them against one another.
  - can do this by running an insertion sort with an increment of 1.
- Generally, shellsort uses a sequence of increments h_t, ...h_1 with the last, h_1, being 1.

for k from t downto 1 do

  for d from 0 to h_k - 1 do

    Insertion sort the sequence A[d], A[d+h_k], A[d+ 2h_k]...

- Last line expands into a doubly nested loop like insertion sort.
  - Can perform the insertion sort on all interlaced sequences simultaneously - you figure that algorithm out.

Shell sort sequences

- As long as last increment is 1, shell sort will work
  - The last iteration is a complete insertion sort - the point of the previous iterations is to reduce the work done in the last iteration.
- Early increments should be larger than the later ones
  - Elements will move longer distances in shorter sequences, reducing the work of later stages where the sequences are longer
- Early increments should not be multiples of later increments
  - Some of the comparisons made with the later increments will be redundant - i.e., we will have sorted subsequences.

Shell sort sequences

- There should not be too many increments
  - If you use O(n) increments, then the entire algorithm will be $\Omega(n^2)$, which is not the goal.
- There must not be too few increments
  - If there are only a constant number of increments, independent of n, then the analysis is similar to regular insertion sort and the algorithm is again quadratic
- A good increment sequence is 1, 4, 13, 40, 121 - generally $H_i = (3^{i} {}^{-1})/2$ (!!!!!)
- Experimentally this gives an algorithm close to O(nlogn)
- But nobody knows the best increment sequence, and shell sort is very hard to analyze mathematically

Shell sort example

11 10 9 8 7 6 5 4 3 2 1 0 **initial sequence**

11 6 1
10 5 0
9 4
8 3
7 2

initial subsequences

1 6 11
0 5 10
4 9
3 8
2 7

sorted subsequences

Shell sort example

1 6 11
0 5 10
4 9
3 8
2 7

1 0 4 3 2 6 5 9 8 7 11 10

At this point, no element is further than 2 positions from its final position in the sorted list - just run insertion sort to finish

Distribution counting - the road to radix sort

- If we knew how many times each item occurred in the input (including those that occur 0 times) then we could compute the output location of any item and avoid all of the move operations
  - This can only be done if the number of possible keys is very small
- Identify a counter, c[i] with every possible key.
  - Make one pass over the input list to count the number of times each k occurs.
  - Make one pass over the counters to determine where to place items in the output list.
  - As an item is placed into the output list, its counter is decremented

Distribution counting - example

Input list:

9a 4a 6a 4b 2a 9b 5a 7a 6b 4c

Possible keys are 0-9

0	1	2	3	4	5	6	7	8	9
2a	4a	4b	4c	5a	6a	6b	7a	9a	9b

Key	Count	Start
0	0	0
1	0	0
2	1	0
3	0	1
4	3	1
5	1	4
6	2	5
7	1	7
8	0	8
9	2	8

Radix sort

- Distribution counting is not practical when the keys occupy a large range of possible values
  - Example - suppose keys are 4 character strings over 8 bit characters
  - This would yield 256*256*256*256 possible 4 character strings, or a 4GB table.
- Keys made of two characters can be sorted by:
  - First, doing a distribution sort of all the keys using the last character as the key value.
  - Then, doing a distribution sort of the resulting table using the **first** character as the key value. The algorithm must be stable so that the relative order of two keys with the same first character but different second characters is preserved

Radix sort

- If we have keys with k characters, radix sort would involve k passes through the table, each pass stably sorting on the next **least** significant character in the key.
- How can this possibly work?

x1 x2 z1 y1 z2 x3 y2

CX AX BZ BY AZ BX AY

SORT ON SECOND CHARACTER USING DISTRIBUTION SORT

c1 a1 b1 b2 a2 b3 a3

CX AX BX BY AY BZ AZ

STABLE SORT ON FIRST CHARACTER WILL YIELD

AX AY AZ BX BY BZ CX

- If two keys have different first characters, they are in the right order because they were put into separate buckets by the second distribution count.
- If two keys have the same first character, then by **stability** step 2 does not change their relative order - but this order was correct after step 1 which sorted on the second character!

Quicksort - road to radix exchange sort

- Partition the input list into two sublists vased on a pivot value, v.
  - a) all elements in the first sublist have values ≤ v
  - b) all elements in the second sublist have values > v.
- Apply this partitioning procedure recursively until all lists are one element long (or sufficiently small that you will sort them using a simple, brute force sort).
- Partitioning is the key step

Partitioning in quicksort

1. Select a position at random in the array - let x be the value chosen
2. Scan the array from left to right until finding an element a[i] with a[i] ≥ x.
3. Scan the array from right to left until finding an element a[j] with a[j] ≤ x
4. Exchange these two elements and continue ths scan and swap until the scans cross - i.e., i > j.

Example of partitioning

22	35	11	42	9	53	23	18	25	7	Partition on 23
i									j	
22	35	11	42	9	53	23	18	25	7	Exchange
22	7	11	42	9	53	23	18	25	35	

Sorting - 17

Example

22	7	11	42	9	53	23	18	25	35	
22	7	11	42	9	53	23	18	25	35	Exchange
22	7	11	18	9	53	23	42	25	35	
22	7	11	18	9	53	23	42	25	35	Exchange
					i	j				

Sorting - 18

Example

22	7	11	18	9	53	23	42	25	35

i j

22	7	11	18	9	23	53	42	25	35

i j

22	7	11	18	9	23	53	42	25	35

Terminate
[1,j] and [i,10]
are ready for
recursive partition

Notice that 23 is in its final position,
so recursive call only has to apply
quicksort to [1, j-1]. This will not
always happen unless we are clever

j i

Example

22	35	11	42	9	53	23	18	25	7

Partition on 22

22	7	11	18	9	53	23	42	25	35

j i

Observations

- A bad choice of pivot can make quicksort very inefficient
  - if we always chose the smallest element, then quicksort is like selection sort
  - can choose a small set of values (3) and take the median
- Recursive nature of quicksort requires a stack for bookkeeping
  - stack may be as large as the input list in the worst case, because we can have up to n recursive calls
  - more efficient to maintain your own stack of intervals to be partitioned

Radix-exchange sort

- Sort on the basis of bits - fixed quicksort partitions
  - first iteration, partition list into two sublists
  - one includes all keys whose first bit is 1
  - other includes all keys whose first bit is 0
  - i'th iteration partitions lists on i'th bits
- Algorithm terminates when there are no bits left to test, or all sublists contain only one element.
- Special case of quicksort with data independent pivot selection

Pivot selection

- Assume each value to be sorted is represented by an M bit number - b_{M-1}, b_{M-2}, ...b_0 .
- After the i'th instantiation of the algorithm, we construct pivots for the sublists created by adding or subtracting 2^{M-i} to the previous pivot
  - For the first sublist we subtract
  - For the second sublist we add
- An example might make things clear!

Example of radix exchange sort

Number	Binary representation
113	0001110001
400	0110010000
818	1100110010
612	1001100100
411	0110011011
311	0100110111
412	0110011100
420	0110100100

start	[113	400	818	612	411	311	412	420]
Bit 9	[113	400	420	412	411	311] |	[612	818]
Bit 8	[113] |	[400	420	412	411	311] |	[612] |	[818]
Bit 7		[311]	[420	412	411	400]		
Bit 6			[420	412	411	400]		
Bit 5			[400	412	411]	[420]		
Bit 4			[400	412	411]			
Bit 3			[400]	[412	411]			
Bit 2				[411]	[412]			