Sorting Algorithms & Binary Search

CSE 2320 – Algorithms and Data Structures

Alexandra Stefan

University of Texas at Arlington

Last updated 8/27/2018

Summary

- Properties of sorting algorithms
- Sorting algorithms
  - Selection sort – Chapter 6.2 (Sedgewick).
  - Insertion sort – Chapter 2 (CLRS, Sedgewick)
    - Pseudocode conventions.
  - Merge sort – CLRS, Chapter 2
- Indirect sorting - (Sedgewick Ch. 6.8 'Index and Pointer Sorting')
- Binary Search
  - See the notation conventions (e.g. $\log_2 N = \lg N$)
- Terminology and notation:
  - $\log_2 N = \lg N$
  - Use interchangeably:
    - *Runtime* and *time complexity*
    - *Record* and *item*

Sorting

Sorting

- Sort an array, **A**, of items (numbers, strings, etc.).
- Why sort it?
  - To use in binary search.
  - To compute rankings, statistics (top-10, top-100, median).
- Study several sorting algorithms,
  - Pros/cons, behavior .
- Today: selection sort, insertion sort.

Properties of sorting

Sedgewick 6.1

- Stable:
  - It does not change the relative order of items whose keys are equal.

- Adaptive:
  - The time complexity will depend on the input
    - E.g. if the input data is almost sorted, it will run significantly faster than if not sorted.
    - see later insertion sort vs selection sort.

Other aspects of sorting

- Time complexity: worst/best/average
- Number of data moves: copy/swap the items
- Space complexity: Extra Memory used
  - *Θ(1): In place* methods: constant extra memory
  - *Θ(N):* Uses extra space proportional top the number of items:
    - For pointers (e.g. linked lists or indirect access)
    - For a copy of the data
- Direct vs indirect sorting
  - Direct: move items as needed to sort
  - Indirect: move *pointers/handles* to items.
    - Can keep the key with pointer or not.
- Later on: non-comparison sorting

Stable sorting

- An item consists of an int (e.g. GPA) and a string (e.g. name).
- Sort based on: GPA (integer)

4 Bob	3 Tom	4 Anna	3 Jane	1 Henry

- Stable sort (OK: Tom before Jane and Bob before Anna):

1 Henry	3 Tom	3 Jane	*4* *Bob*	*4* *Anna*

- Unstable sort (violation: Anna is now before Bob):

1 Henry	3 Tom	3 Jane	4 Anna	4 Bob

- Note: Stable is a property of the algorithm, NOT of one the pair algorithm-data.

Stable sorting

- Applications
  - Sorting by 2 criteria,
    - E.g.: 1st by GPA, 2nd by name:
      - When the GPA is the same, have data in order of names
    - Solution:
      - First sort by name (with any method)
      - Next, with a stable sort, sort by GPA
    - Alternative solution:
      - write a more complex comparison function.
  - Part of other sorting methods
    - See later: LSD radix sort uses a stable sort (count sort).

Proving an Algorithm is Stable

- An algorithm is stable if we can guarantee/prove that this property happens for any input (not just a few example inputs).

  => To prove it, must use an actual proof (possibly using a loop invariant) or give very good explanation. (A few examples are not a proof.)

- An algorithm is not stable if there is at least one possible input for which it breaks the property.

  => To prove it, find one example input for which the property fails.

- Intuition: if an algorithm swaps items that are away from each other (jump over other items) it is likely NOT stable.
  - This statement is a guideline, not a proof. Make sure you always find an example if you suspect this case.

Selection sort

Selection sort vs Insertion sort

Each row shows the array after one iteration of the outer loop for each algorithm.

Selection sort

Select the smallest element from the remaining ones and swap it in place.

original	5	3	7	8	6	0	4
1st	**0**	3	7	8	6	**5**	4
2nd	0	**3**	7	8	6	5	4
3rd	0	3	**4**	8	6	5	**7**
4th	0	3	4	**5**	6	**8**	7
5th	0	3	4	5	**6**	8	7
6th	0	3	4	5	6	**7**	**8**

Insertion sort

Insert the next element in it's place in the sorted sequence to the left of it.

5	3	7	8	6	0	4
3	**5**	7	8	6	0	4
3	5	**7**	8	6	0	4
3	5	7	**8**	6	0	4
3	5	**6**	7	8	0	4
0	3	5	6	7	8	4
0	3	**4**	5	6	7	8

Elements in shaded cells are sorted, but:

- Selection sort: out of the whole array; they are in their final position (see 0 on first cell)
- Insertion sort: out of numbers originally in the shaded cells; not in final position (e.g. see the 8 move all the way to the right).

Selection Sort

- Given unsorted array, **A**.
- From left to right, put the remaining smallest element on it's final position
  - Find the smallest element, and exchange it with element at position 0.
  - Find the second smallest element, and exchange it with element at position 1.
  - …
  - Find the i-th smallest element, and exchange it with element at position i-1.

  - If we do this |**A**|-1 times, then **A** will be sorted.

- Resources:
  - RSI (nice, but it selects the maximum element, not the minimum): http://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html#lst-selectionsortcode
  - Animation: https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Selection Sort – Code
(Sedgewick)

- Variable names renamed as in insertion sort (i <->j).

```
/* sort array A in ascending order.
   N is the number of elements in A. */
void selection(int A[], int N)
{ int i, j, temp;
  for (j = 0; j < N-1; j++)
  {//min_idx: index of the smallest remaining item.
    int min_idx = j;
    for (i = j+1; i < N; i++)
      if (A[i] < A[min_idx]) min_idx = i;
    // swap A[min_idx] <-> A[j]
    temp = A[min_idx];
    A[min_idx] = A[j];
    A[j] = temp;
  }
}
```

Selection Sort

```
/* Based on code from Sedgewick (renamed i<->j)
   Sort array A in ascending order.
   N is the number of elements in A. */
void selection(int A[], int N)
{ int i, j, temp;
  for (j = 0; j < N-1; j++)
  {//min_idx: index of the smallest remaining item.
    int min_idx = j;
    for (i = j+1; i < N; i++)
      if (A[i] < A[min_idx]) min_idx = i;
    // swap A[min_idx] <-> A[j]
    temp = A[min_idx];
    A[min_idx] = A[j];
    A[j] = temp;
  }
}
```

- Is it stable?
- How much extra memory does it need?
- Does it access the data directly?
- Is it adaptive?
  - What time complexity does it have in best, worst, average cases?
- Number of data moves?

Selection Sort Properties

- Is it stable?
  - No. Problem: swaps (The problem is not the fact that the smallest number jumps to index 0, but that number at index 0 may jump over another item of the same value.).
    - E.g. (4,Jane), (4,Anna), (1, Sam) ---> (1, Sam), (4,Anna), (4,Jane)
- How much extra memory does it require?
  - Constant (does not depend on input size N)
- Does it access the data directly?
  - Yes (as given).
  - Exercise: Modify selection sort to access the data indirectly.
- Is it adaptive?
  - No. The algorithm will do the SAME number of comparisons and the same number of swaps (even when the data is sorted or in any other special order).
- Number of data moves?
  - Linear: 3N (contrast this with the quadratic time complexity)

Selection Sort – Time Analysis Worksheet

```
void selection(int A[], int N)
{ int i, j, temp;
  for (j = 0; j < N-1; j++)
  { int min_idx = j;
    for (i = j+1; i < N; i++)
      if (A[i] < A[min_idx])
          min_idx = i;
    temp = A[min_idx];
    A[min_idx] = A[j];
    A[j] = temp;
  } //end of j loop
} //end of method
```

iterations → ____

iterations → ____

j	Inner loop started = N-1-j
0	
1	
2	
…	
N-3	
N-2	

Total times the inner loop started (for all values of j):
T(N) =

Selection Sort – Time Analysis Solution

```
void selection(int A[], int N)
{ int i, j, temp;
  for (j = 0; j < N-1; j++)
  { int min_idx = j;
    for (i = j+1; i < N; i++)
      if (A[i] < A[min_idx])
          min_idx = i;
    temp = A[min_idx];
    A[min_idx] = A[j];
    A[j] = temp;
  } //end of j loop
} //end of method
```

iterations → (N-1)

iterations → (N-1-j)

(depends on j)

j	Inner loop started = N-1-j
0	N-1
1	N-2
2	N-3
…	…
N-3	2
N-2	1

Total times the inner loop started (for all values of j):

T(N) = (N-1) + (N-2) + … 2 + 1 =

= [N * (N-1)]/2 -> N^2 order of magnitude

Note that the N^2 came from the summation NOT because 'there is an N in the inner loop' (NOT because N * N).

Selection Sort – Time Analysis (more details)

```
// Assume the inner loop executes 4
// instructions every time it starts.
void selection(int A[], int N)
{ int i, j, temp;
  for (j = 0; j < N-1; j++)
  { int min_idx = j;
    for (i = j+1; i < N; i++)
      if (A[i] < A[min_idx])
          min_idx = i;
    temp = A[min_idx];
    A[min_idx] = A[j];
    A[j] = temp;
  } //end of j loop
} //end of method
```

iterations → (N-1)

iterations → (N-1-j) *4

j	Iterations of inner loop = N-1-j
0	(N-1) *4
1	(N-2) *4
2	(N-3) *4
…	…
N-3	2*4
N-2	1*4

Total instructions executed in the inner loop (over all values of j):
T(N) = (N-1) *4 + (N-2) *4 + … 2*4 + 1*4 =
= 4*[N * (N-1)]/2 -> still N^2 order of magnitude
Adding other details such as the exact count of instructions for the outer loop would add other lower order terms (N and constants).

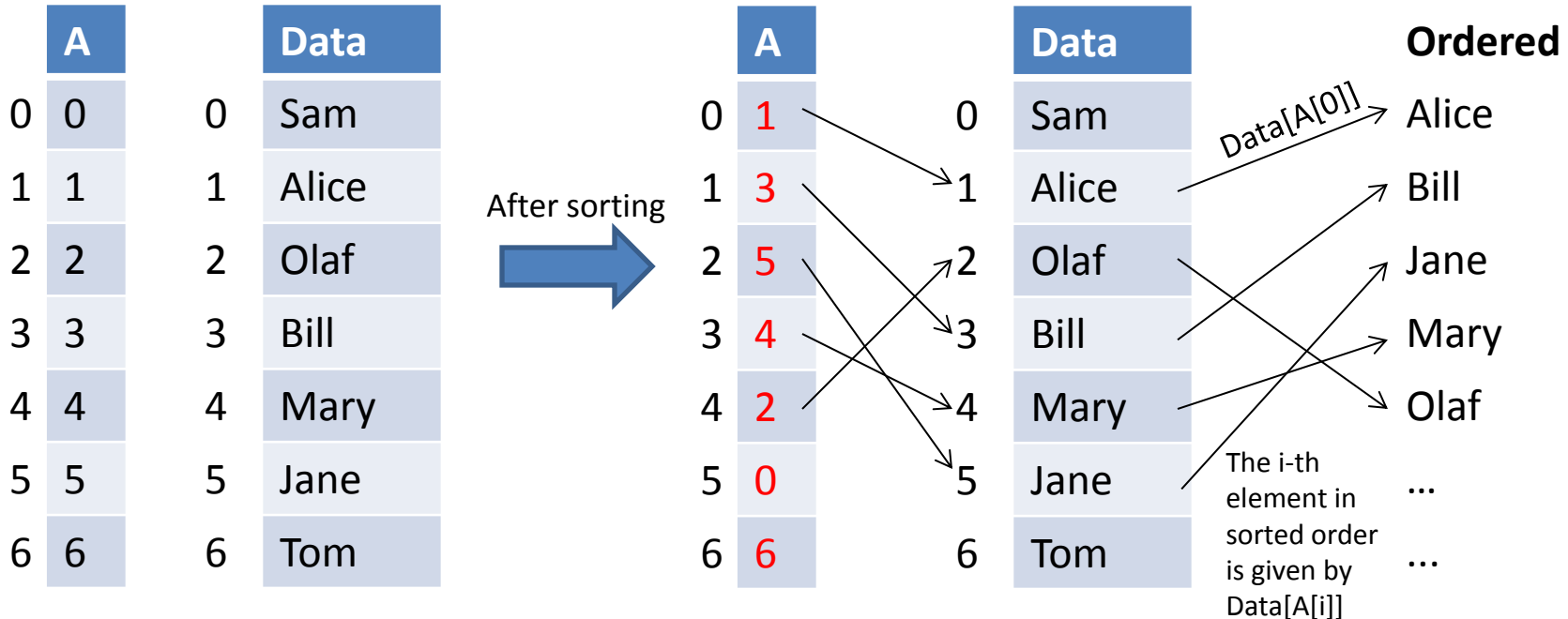
Selection Sort - Time Analysis

- Step 1: Find smallest element, and exchange it with element at index 0.
  - N-1 comparisons (and 3 data moves)
- Step 2: Find 2nd smallest element, and exchange it with element at index 1.
  - N-2 comparisons (and 3 data moves)
- …
- Step i: find i-th smallest element, and exchange it with element at index i-1.
  - N-i comparisons (and 3 data moves)
- Total: $(N-1) + (N-2) + (N-3) + \ldots + 1 = [(N-1)*N]/2 = (N^2 - N)/2$
  - about $N^2/2$ comparisons. (dominant term: N^2)
- **Quadratic time complexity**: the dominant term is N^2.
  - Data moves: linear
- Commonly used sorting algorithms have N * lg(N) time complexity, which is much better (as N gets large). (See mergesort)

Indirect Sorting

- If you cannot move the data, e.g. because:
  - either records are too big, or
  - you do not have permission to change the data.
- Solution:
  - In the array A keep indexes to the original data
  - Rearrange the indexes, in A, to give the data in sorted order.

- Indirect selection sort:
  - Takes both the data array and A (with indexes in order 0,…N-1) and rearrange A based on the data.



Indirect Sorting with Selection Sort

```
/* Data - array with N records (the actual data).
   A - array with numbers 0 to N-1 (indexes in Data).
   Rearrange A s.t. when using its items as indexes in
Data, we access the records in Data in increasing order.
*/
void selection_indir(int A[], int N, DataType Data[])
{ int i, j, temp;
  for (j = 0; j < N-1; j++)
  { int min_idx = j;
    for (i = j+1; i < N; i++)
      if (Data[A[i]] < Data[A[min_idx]]) min_idx = i;
    // swap indexes (from A). Do NOT swap data.
    temp = A[min_idx]; A[min_idx] = A[j]; A[j] = temp;
  }
}
```

Indirect Sorting & Binary Search

- Use A (with sorted indexes for Data) to perform binary search in Data.
  - E.g. search for: Xena, Alice, Dan

	A
0	1
1	3
2	5
3	4
4	2
5	0
6	6

	Data
0	Sam
1	Alice
2	Olaf
3	Bill
4	Mary
5	Jane
6	Tom

Pseudocode conventions:

(CLRS Page 20)

- Indentation shows body of loop or of a branch
- y = x treated as pointers so changing x will change y.
- cascade: x.f.g
- NILL used for the NULL pointer
- Pass by value of pointer: if x is a parameter, x=y will not be preserved but x.j=3 will be (when returned back to the caller fct)
- Pseudocode will allow multiple values to be returned with one return statement.
- The Boolean operators “and” and “or” are short circuiting: “x != NILL and x.f!=3 ” is safe.
- “the keyword “error” indicates that an error occurred because the conditions were wrong for the procedure to be called.” - CLRS

Insertion sort

Insertion sort

Process the array from left to right.
Step j (outer loop):
- elements A[0],A[1],…A[j-1] are already sorted
- insert element A[j] in it's place among A[0],..A[j-1] (inner loop)

Each row shows the array after one iteration of the outer loop (after step j).

original	5	3	7	8	6	0	4
1st	**3**	5	7	8	6	0	4
2nd	3	5	**7**	8	6	0	4
3rd	3	5	7	**8**	6	0	4
4th	3	5	**6**	7	8	0	4
5th	**0**	3	5	6	7	8	4
6th	0	3	**4**	5	6	7	8

Elements in shaded cells are sorted, but they have only items that were originally in the shaded cells. They are not in final position (e.g. see the 8 move all the way to the right).

- Brief and nice resource: http://interactivepython.org/runestone/static/pythonds/SortSearch/TheInsertionSort.html
- Animation for Sedgewick's version (insert by swapping with smaller elements to the left): https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

CLRS pseudocode

Pseudocode questions?

- Note lack of details: no types, no specific syntax (C, Java,…)
- But sufficient specifications to implement it: indexes, data updates, arguments, …

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted
           sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Proving That an Algorithm is Correct

- **Required for the exam. Read the relevant book section.**
- See CLRS, (starting at page 18), for proof of loop invariant and correctness of the insertion sort algorithm:
  - Identify a property that is preserved (maintained, built) by the algorithm: "the loop invariant" (b.c. preserved by the loop)
    - Which loop would you use here?
  - Show:
    - Initialization
    - Maintenance
    - Termination – use that property/invariant to show that the algorithm is correct
- What would the loop invariant be for the inner loop for insertion sort?
  - This question may be part of your next homework or quiz.

CLRS pseudocode

Same as previous slide, but indexes start from 0. Changes:
Line 1: j =1
Line 5: (i>=0)

```
Insertion-Sort(A,N)
1.  for j = 1 to N-1
2.     key = A[j]
3.     // insert A[j] in the
       // sorted sequence A[0…j-1]
4.     i = j-1
5.     while (i>=0) and (A[i]>key)
6.        A[i+1] = A[i]
7.        i = i-1
8.     A[i+1] = key
```

Compute time complexity:

Version 1: each instruction has a different cost
(See CLRS, pg 26).

Version 2: all instructions have the same cost.

How many times will the while loop (line 5) execute? Give the best, worst and average cases?

Insertion Sort – Time Complexity Worksheet

- Assume all instructions have cost 1.
- See book for analysis using instruction cost.

```
Insertion-Sort(A,N)
1.  for j = 1 to N-1
2.     key = A[j]
3.     // insert A[j] in the
       // sorted sequence A[0…j-1]
4.     i = j-1
5.     while (i>=0) and (A[i]>key)
6.        A[i+1] = A[i]
7.        i = i-1
8.     A[i+1] = key
```

iterations →

j	Inner loop iterations: Best : 1	Worst: j	Average: j/2
1			
2			
…			
N-2			
N-1			

iterations →

At most:

At least:

Insertion Sort – Time Complexity Solution

- Assume all instructions have cost 1.
- See book for analysis using instruction cost.

```
Insertion-Sort(A,N)
1.  for j = 1 to N-1
2.     key = A[j]
3.     // insert A[j] in the
       // sorted sequence A[0…j-1]
4.     i = j-1
5.     while (i>=0) and (A[i]>key)
6.        A[i+1] = A[i]
7.        i = i-1
8.     A[i+1] = key
```

iterations → **N-1**

j	Inner loop iterations: Best : 1	Worst: j	Average: j/2
1	1	1	1/2
2	1	2	2/2
…	…		
N-2	1	N-2	(N-2)/2
N-1	1	N-1	(N-1)/2

iterations →

At most: j

Includes end loop check

At least: 1

Evaluate the condition: (i>0 and A[i]>key)

Insertion Sort Time Complexity Cont.

j	Inner loop iterations: Best : 1	Worst: j	Average: j/2
1	1	1	1/2
2	1	2	2/2
…	…		
N-2	1	N-2	(N-2)/2
N-1	1	N-1	(N-1)/2
Total			
Order of magnitude			
Data that produces it.			

'Total' instructions in worst case math derivation:
T(N) =

Order of magnitude:

Insertion Sort Time Complexity Cont.

j	Inner loop iterations: Best : 1	Worst: j	Average: j/2
1	1	1	1/2
2	1	2	2/2
…	…		
N-2	1	N-2	(N-2)/2
N-1	1	N-1	(N-1)/2
Total	(N-1)	[N * (N-1)]/2	[N * (N-1)]/4
Order of magnitude	N	N^2	N^2
Data that produces it.	Sorted	Sorted in reverse order	Random data

=> $O(N^2)$

'Total' instructions in worst case:

T(N) = (N-1) + (N-2) + … 2 + 1 =

= [N * (N-1)]/2 -> N^2 order of magnitude

Note that the N^2 came from the summation, NOT because 'there is an N in the inner loop' (NOT because N * N).

See the Khan Academy for a discussion on the use of O(N2): https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort

Time complexity

- For what inputs do we achieve these behaviors:
  - Best
  - Worst
  - Average

- Insertion sort is adaptive:
  - If **A** is sorted, the runtime is proportional to **N**.

Insertion sort - Properties

- Is this particular ‘implementation’ stable?
- Note how an algorithm has the capability to be stable but the way it is implemented can still make it unstable.
  - What happens if we use *A[i]>=key* in line 5?
- Give an implementation that uses a sentinel (to avoid the *i>0* check in line 5)
  - What is a sentinel?
  - Is it still stable? (How do you update/move the sentinel)?
  - Time complexity trade-off:
    - Cost to set-up the sentinel (linear) vs
    - Savings from removing the i>0 check (quadratic, in worst case).

Comparisons vs Swaps

- Compare the data comparisons and data movement performed by selection sort and insertion sort.

Selection sort & Insertion sort

Time Complexity

Each row shows the array after one iteration of the outer loop for each algorithm.

Selection sort

Select the smallest element from the remaining ones and swap it in place.

original	5	3	7	8	6	0	4
1st	**0**	3	7	8	6	**5**	4
2nd	0	**3**	7	8	6	5	4
3rd	0	3	**4**	8	6	5	**7**
4th	0	3	4	**5**	6	**8**	7
5th	0	3	4	5	**6**	8	7
6th	0	3	4	5	6	**7**	**8**

White cells are visited by the iterations of the inner loop => they are proportional with the time complexity => ~ $N^2/2$

Insertion sort

Insert the next element in it's place in the sorted sequence to the left of it.

5	3	7	8	6	0	4
3	**5**	7	8	6	0	4
3	5	**7**	8	6	0	4
3	5	7	**8**	6	0	4
3	5	**6**	7	8	0	4
0	3	5	6	7	8	4
0	3	**4**	5	6	7	8

Gray cells are visited by the iterations of the inner loop => they are proportional with the time complexity => ~ $N^2/2$

Insertion sort
Data Movement

'Data movement' is an assignment.
(Implementation will matter: deep copy or pointer)

```
Insertion-Sort(A,N)
1.  for j = 1 to N-1
2.     key = A[j]
3.     // insert A[j] in the
       // sorted sequence A[0…j-1]
4.     i = j-1
5.     while (i>=0) and (A[i]>key)
6.        A[i+1] = A[i]
7.        i = i-1
8.     A[i+1] = key
```

5	3	7	8	6	0	4
3	**5**	7	8	6	0	4
3	5	**7**	8	6	0	4
3	5	7	**8**	6	0	4
3	5	**6**	**7**	**8**	0	4
0	**3**	**5**	**6**	**7**	**8**	4
0	3	**4**	**5**	**6**	**7**	**8**

Each red number: 2 moves.
Each blue number: 1 move.
Best:
Worst: Average:

Selection sort & Insertion sort

Data Movement

Each row shows the array after one iteration of the outer loop for each algorithm.

Selection sort

Select the smallest element from the remaining ones and swap it in place.

original	5	3	7	8	6	0	4
1st	**0**	3	7	8	6	**5**	4
2nd	0	**3**	7	8	6	5	4
3rd	0	3	**4**	8	6	5	**7**
4th	0	3	4	**5**	6	**8**	7
5th	0	3	4	5	**6**	8	7
6th	0	3	4	5	6	**7**	**8**

Each red pair: 3 data moves.

Best: Worst:

Average:

Insertion sort

Insert the next element in it's place in the sorted sequence to the left of it.

5	3	7	8	6	0	4
3	**5**	7	8	6	0	4
3	5	**7**	8	6	0	4
3	5	7	**8**	6	0	4
3	5	**6**	**7**	**8**	0	4
0	**3**	**5**	**6**	**7**	**8**	4
0	3	**4**	**5**	**6**	**7**	**8**

Each red number: 2 moves.
Each blue number: 1 move.

Best: Worst:

Average:

Selection sort & Insertion sort

Data Movement - Answer

Each row shows the array after one iteration of the outer loop for each algorithm.

Selection sort

Select the smallest element from the remaining ones and swap it in place.

original	5	3	7	8	6	0	4
1st	**0**	3	7	8	6	**5**	4
2nd	0	**3**	7	8	6	5	4
3rd	0	3	**4**	8	6	5	**7**
4th	0	3	4	**5**	6	**8**	7
5th	0	3	4	5	**6**	8	7
6th	0	3	4	5	6	**7**	**8**

Each red pair: 3 data moves.

Best: 3(N-1) Worst: 3(N-1)

Average: 3(N-1) (not adaptive)

Insertion sort

Insert the next element in it's place in the sorted sequence to the left of it.

5	3	7	8	6	0	4
3	**5**	7	8	6	0	4
3	5	**7**	8	6	0	4
3	5	7	**8**	6	0	4
3	5	**6**	**7**	**8**	0	4
0	**3**	**5**	**6**	**7**	**8**	4
0	3	**4**	**5**	**6**	**7**	**8**

Each red number: 2 moves.

Each blue number: 1 move.

Best: 2N Worst: 2N+N(N-1)/2

Average:2N+N(N-1)/4

Practice

- Assume you have a computer that executes 10^4 instructions per second. You run on it selection sort and insertion sort for 10^9 large records and it takes 1 second to copy one record. Fill in the table below:

	Time (in sec) due to comparisons	Time (in sec) due to data moves
Insertion sort (average case)		
Selection sort		

Binary Search

- Given set **A** of **N** objects.
- Assume objects in **A** are *sorted in ascending* order.
- Given object **v**, determine if **v** is in **A**.
- For simplicity, now objects are **integers**.
  - The solution works for much more general types of objects.

Binary Search

- Initially:
  - **left** = **0**
  - **right** = **N – 1**
  - **m = (left+right)/2** (middle index, between left and right)
- Repeat
  - Compare **v** with **A[m]**.
    - If **v == A[m]**, we found **v**, so we are done (break).
    - If **v < A[m]**, then **right = m - 1**.
    - If **v > A[m]**, then **left = m + 1**.
- We have reduced our search range in half, with a few instructions.
- For more flexibility `left` and `right` can be given as arguments to the function:

```
int search(int A[], int N, int v, int left, int right)
```

- See animation: https://www.cs.usfca.edu/~galles/visualization/Search.html
  - The array stretches on 2 or more lines

Binary search – code

- For convenience

```
/* code from Sedgewick
   Determines if v is an element of A.
   If yes, returns the position of v in A.
   If not, returns -1.
   N is the size of A.*/

1. int search(int A[], int N, int v){
2.   int left, right;
3.   left = 0; right = N-1;
4.   while (left <= right)
5.   { int m = (left+right)/2;
6.     if (v == A[m]) return m;
7.     if (v < A[m])
8.         right = m-1;
9.     else
10.      left = m+1;
11.  }
12.  return -1;
13. }
```

Binary Search - Example

Search for **Dan** in sorted array.

v = ...

left	right	middle	Action (comparison)

0	Alice
1	Cathy
2	Emma
3	John
4	Paul
5	Sam
6	Tom

```
/* code from Sedgewick
   Determines if v is an element of A.
   If yes, returns the position of v in A.
   If not, returns -1.
   N is the size of A.*/

1. int search(int A[], int N, int v){
2.    int left, right;
3.    left = 0; right = N-1;
4.    while (left <= right)
5.    { int m = (left+right)/2;
6.      if (v == A[m]) return m;
7.      if (v < A[m])
8.          right = m-1;
9.      else
10.       left = m+1;
11.   }
12.   return -1;
13. }
```

Binary Search - Example

Search for Dan in sorted array:

(Searching for Emma results in visiting the same items.)

left	right	middle	Action (comparison)
0	6	3	Dan < John (go left)
0	2	1	Dan > Cathy (go right)
2	2	2	Dan < Emma (go left)
2	1		Indexes cross, stop. Not found

0	Alice
1	Cathy
2	Emma
3	John
4	Paul
5	Sam
6	Tom

```
/* code from Sedgewick
   Determines if v is an element of A.
   If yes, returns the position of v in A.
   If not, returns -1.
   N is the size of A.*/

1. int search(int A[], int N, int v){
2.    int left, right;
3.    left = 0; right = N-1;
4.    while (left <= right)
5.    { int m = (left+right)/2;
6.      if (v == A[m]) return m;
7.      if (v < A[m])
8.          right = m-1;
9.      else
10.       left = m+1;
11.   }
12.   return -1;
13. }
```

Binary Search - Code

```
/* code from Sedgewick
   Determines if v is an element of A.
   If yes, returns the position of v in A.
   If not, returns -1.
   N is the size of A.*/

1. int search(int A[], int N, int v){
2.   int left, right;
3.   left = 0; right = N-1;
4.   while (left <= right)
5.   { int m = (left+right)/2;
6.     if (v == A[m]) return m;
7.     if (v < A[m])
8.        right = m-1;
9.     else
10.       left = m+1;
11.  }
12.  return -1;
13. }
```

1. How many times will the while loop repeat for N = 32 ?

Best case: 1 iteration (less) (found at first comparison)

Worst case: 6 iterations (when not found, or found last)

Candidates (=right–left+1):

32

16

8

4

2

1

0 (stop, indexes cross over)

=> 6 iterations => $\lfloor \lg N \rfloor + 1$

Verify with other cases, e.g. N: 17, 31, 33

Deriving and Verifying the Formula

- Max new candidates at next iteration: $\left\lceil \frac{N-1}{2} \right\rceil$
  - (N-1) b.c. middle position is removed
- Examples verifying the formula: $\lfloor \lg N \rfloor + 1$

N (==2^5)
$32 = 2^5$
$16 = 2^4$
$8 = 2^3$
$4 = 2^2$
$2 = 2^1$
$1 = 2^0$
5+1 = 6 iterations
0 – stop, not counted

When N is an exact power, e.g. $N = 2^p$, then we have $(p+1)$ iterations.

N (<2^5):
17
8
4
2
1
0 – stop
5 iter = 4 +1

N (<2^5):
31
15
7
3
1
0 – stop
5 iter = 4 +1

When N is NOT an exact power, e.g. $2^{p-1} < N < 2^p$, then we have p iterations.

Time Analysis of Binary Search

- How many elements do we need to compare **v** with, if **A** contains **N** objects?
  - At most $\log_2(N)$.
- We call it **logarithmic time complexity**.
- Fast!

Extra Materials
(Not Required)

Insertion sort code (Sedgewick)

```
void insertion(int A[], int left, int right)
  { int i;
    // place the smallest item on the leftmost position: sentinel
    for (i = left+1; i <= right; i++)
       compexch(A[left], A[i]);
    for (i = left+2; i <= right; i++)
      { int j = i; Item v = A[i];
        while (less(v, A[j-1]))
          { A[j] = A[j-1]; j--; }
        A[j] = v;
      }
  }
```

Binary Search - Recursive

```
/* Adapted from Sedgewick
*/
int search(int A[], int left, int right, int v)
  { int m = (left+right)/2;
    if (left > right) return -1;
    if (v == A[m]) return m;
    if (left == right) return -1;
    if (v < A[m])
         return search(A, left, m-1, v);  // recursive call
    else
          return search(A, m+1, right, v);  // recursive call

  }
```

- How many recursive calls?
- See the correspondence between this and the iterative version.