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
  – $\Theta(1)$: *In place* methods: constant extra memory
  – $\Theta(N)$: Uses extra space proportional to 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)

<table>
<thead>
<tr>
<th>GPA</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Bob</td>
</tr>
<tr>
<td>3</td>
<td>Tom</td>
</tr>
<tr>
<td>4</td>
<td>Anna</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
</tr>
<tr>
<td>1</td>
<td>Henry</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Henry</th>
<th>Tom</th>
<th>Jane</th>
<th>Bob</th>
<th>Anna</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Henry</td>
<td>Tom</td>
<td>Jane</td>
<td>Bob</td>
<td>Anna</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Henry</th>
<th>Tom</th>
<th>Jane</th>
<th>Anna</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Henry</td>
<td>Tom</td>
<td>Jane</td>
<td>Anna</td>
<td>Bob</td>
</tr>
</tbody>
</table>

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

• Applications
  – Sorting by 2 criteria,
    • E.g.: 1\textsuperscript{st} by GPA, 2\textsuperscript{nd} 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.

<table>
<thead>
<tr>
<th>original</th>
<th>5</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>6</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>2nd</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3rd</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>4th</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>5th</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>6th</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

#### Insertion sort

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

<table>
<thead>
<tr>
<th>original</th>
<th>5</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>6</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2nd</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3rd</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4th</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>5th</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6th</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

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
• 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;
        temp = A[min_idx];
        A[min_idx] = A[j];
        A[j] = temp;
    }
}
/* 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;
            temp = A[min_idx];
            A[min_idx] = A[j];
            A[j] = temp;
        }
}
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)
void selection(int A[], int N)
{
    int i, j, temp;
    for (j = 0; j < N-1; j++) { //iterations
        int min_idx = j;
        for (i = j+1; i < N; i++) { //iterations
            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
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;
    }
}

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).
// 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;
    }
}

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: \((N-1) + (N-2) + (N-3) + \ldots + 1 = \left\lfloor (N-1)\frac{N}{2} \right\rfloor = \frac{N^2-N}{2}\)
  - 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 \(N \times \log(N)\) time complexity, which is much better (as \(N\) gets large). (See mergesort)
Indirect Sorting

- **Indirect sorting**: does not modify the original data in any way.
- **Applications**:
  - records are too big to move around (it takes too long to copy a record), or
  - you do not have permission to change anything in the original collection (array, list, ...).
- **Solution**:
  - In the array A keep indexes to the original data
  - Rearrange the indexes, in A, to give access to the records 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 comparisons among the Data records. The Data array was not modified.
  - What would it do for a linked list?

<table>
<thead>
<tr>
<th>A</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 Sam</td>
</tr>
<tr>
<td>1</td>
<td>1 Alice</td>
</tr>
<tr>
<td>2</td>
<td>2 Olaf</td>
</tr>
<tr>
<td>3</td>
<td>3 Bill</td>
</tr>
<tr>
<td>4</td>
<td>4 Mary</td>
</tr>
<tr>
<td>5</td>
<td>5 Jane</td>
</tr>
<tr>
<td>6</td>
<td>6 Tom</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 Alice</td>
</tr>
<tr>
<td>1</td>
<td>3 Bill</td>
</tr>
<tr>
<td>2</td>
<td>5 Olaf</td>
</tr>
<tr>
<td>3</td>
<td>4 Jane</td>
</tr>
<tr>
<td>4</td>
<td>2 Mary</td>
</tr>
<tr>
<td>5</td>
<td>0 Jane</td>
</tr>
<tr>
<td>6</td>
<td>6 Tom</td>
</tr>
</tbody>
</table>

After sorting, the i-th element in sorted order is given by Data[A[i]].
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.
    }
}
Indirect Sorting & Binary Search

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

<table>
<thead>
<tr>
<th>A</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sam</td>
</tr>
<tr>
<td>1</td>
<td>Alice</td>
</tr>
<tr>
<td>2</td>
<td>Olaf</td>
</tr>
<tr>
<td>3</td>
<td>Bill</td>
</tr>
<tr>
<td>4</td>
<td>Mary</td>
</tr>
<tr>
<td>5</td>
<td>Jane</td>
</tr>
<tr>
<td>6</td>
<td>Tom</td>
</tr>
</tbody>
</table>
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 its place among $A[0], ..., A[j-1]$ (inner loop)

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

<table>
<thead>
<tr>
<th>original</th>
<th>5</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>6</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1\textsuperscript{st}</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2\textsuperscript{nd}</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3\textsuperscript{rd}</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4\textsuperscript{th}</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>5\textsuperscript{th}</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6\textsuperscript{th}</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

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](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](https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html)
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.
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(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)
7. i = i-1
8. A[i+1] = key
## 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. \hspace{200pt} key = A[j]
3. \hspace{200pt} // insert $A[j]$ in the
   \hspace{200pt} // sorted sequence $A[0...j-1]$
4. \hspace{200pt} $i = j-1$
5. \hspace{200pt} while ($i >= 0$) and ($A[i] > key$)
6. \hspace{200pt} \hspace{200pt} $A[i+1] = A[i]$
7. \hspace{200pt} \hspace{200pt} $i = i-1$
8. \hspace{200pt} \hspace{200pt} $A[i+1] = key$

<table>
<thead>
<tr>
<th>$j$</th>
<th>Inner loop iterations:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best :</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>j</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>N-2</td>
<td></td>
</tr>
<tr>
<td>N-1</td>
<td></td>
</tr>
</tbody>
</table>

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)
7.     i = i-1
8.   A[i+1] = key

Inner loop iterations:

<table>
<thead>
<tr>
<th>j</th>
<th>Best</th>
<th>Worst</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>j</td>
<td>j/2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2/2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-2</td>
<td>1</td>
<td>N-2</td>
<td>(N-2)/2</td>
</tr>
<tr>
<td>N-1</td>
<td>1</td>
<td>N-1</td>
<td>(N-1)/2</td>
</tr>
</tbody>
</table>

At most: j
Includes end loop check
At least: 1
Evaluate the condition: (i>0 and A[i]>key)
## Insertion Sort Time Complexity Cont.

<table>
<thead>
<tr>
<th>j</th>
<th>Inner loop iterations:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best : 1</td>
<td>Worst : j</td>
<td>Average : j/2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1/2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2/2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-2</td>
<td>1</td>
<td>N-2</td>
<td>(N-2)/2</td>
<td></td>
</tr>
<tr>
<td>N-1</td>
<td>1</td>
<td>N-1</td>
<td>(N-1)/2</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Order of magnitude</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data that produces it.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

‘Total’ instructions in worst case math derivation:

\[ T(N) = \]

Order of magnitude:
Insertion Sort Time Complexity Cont.

<table>
<thead>
<tr>
<th>j</th>
<th>Inner loop iterations:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best: 1</td>
<td>Worst: j</td>
<td>Average: j/2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1/2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2/2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-2</td>
<td>1</td>
<td>N-2</td>
<td>(N-2)/2</td>
</tr>
<tr>
<td>N-1</td>
<td>1</td>
<td>N-1</td>
<td>(N-1)/2</td>
</tr>
<tr>
<td>Total</td>
<td>(N-1)</td>
<td>[N * (N-1)]/2</td>
<td>[N * (N-1)]/4</td>
</tr>
</tbody>
</table>

Order of magnitude

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>N²</th>
<th>N²</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data that produces it.</td>
<td>Sorted</td>
<td>Sorted in reverse order</td>
<td>Random data</td>
</tr>
</tbody>
</table>

Total instructions in worst case:

\[ T(N) = (N-1) + (N-2) + ... + 2 + 1 = \]

\[ = [N * (N-1)]/2 \rightarrow N^2 \text{ order of magnitude} \]

Note that the \( N^2 \) came from the summation, NOT because ‘there is an \( N \) in the inner loop’ (NOT because \( N \times N \)).

See the Khan Academy for a discussion on the use of \( \Theta(N^2) \):
[https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort](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 $A[i] \geq key$ 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.

<table>
<thead>
<tr>
<th>original</th>
<th>5</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>6</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3&lt;sup&gt;rd&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>4&lt;sup&gt;th&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>5&lt;sup&gt;th&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>6&lt;sup&gt;th&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

**White** cells are visited by the iterations of the inner loop => they are proportional with the time complexity => \( \sim \frac{N^2}{2} \)

### Insertion sort

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

<table>
<thead>
<tr>
<th>5</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>6</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

**Gray** cells are visited by the iterations of the inner loop => they are proportional with the time complexity => \( \sim \frac{N^2}{2} \)
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)
7. i = i-1
8. A[i+1] = key

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>6</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>original</th>
<th>5</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>6</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3&lt;sup&gt;rd&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>4&lt;sup&gt;th&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>5&lt;sup&gt;th&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>6&lt;sup&gt;th&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

- 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.

<table>
<thead>
<tr>
<th>original</th>
<th>5</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>6</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt;</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3&lt;sup&gt;rd&lt;/sup&gt;</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4&lt;sup&gt;th&lt;/sup&gt;</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>5&lt;sup&gt;th&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6&lt;sup&gt;th&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

- 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.

<table>
<thead>
<tr>
<th>original</th>
<th>5</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>6</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3&lt;sup&gt;rd&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>4&lt;sup&gt;th&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>5&lt;sup&gt;th&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>6&lt;sup&gt;th&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>original</th>
<th>5</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>6</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt;</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3&lt;sup&gt;rd&lt;/sup&gt;</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4&lt;sup&gt;th&lt;/sup&gt;</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5&lt;sup&gt;th&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6&lt;sup&gt;th&lt;/sup&gt;</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

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 $10^{10}$ instructions per second. You run on it selection sort and insertion sort for $10^6$ large records and it takes 1 milisecond to copy one record. Fill in the table below:

<table>
<thead>
<tr>
<th></th>
<th>Time (in sec) due to comparisons</th>
<th>Time (in sec) due to data moves</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(average case)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selection sort</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time (in sec) due to comparisons:
- Insertion sort (average case): $\frac{10^6 \times 1}{10^{10}} \times 1 = 10^{-4}$ seconds
- Selection sort: $\frac{10^6 \times 1}{10^{10}} \times 1 = 10^{-4}$ seconds

Time (in sec) due to data moves:
- Insertion sort (average case): $\frac{10^6 \times 1}{10^{10}} \times 0.001 = 10^{-4}$ seconds
- Selection sort: $\frac{10^6 \times 1}{10^{10}} \times 0.001 = 10^{-4}$ seconds
Binary Search

• Problem:
  – Given object \( v \), determine if \( v \) is in array \( A \).
  – Assume \( A \) has \( N \) objects sorted in ascending order.

• Algorithm:
  – Initially:
    • \( \text{left} = 0 \), \( \text{right} = N - 1 \)
    • \( m = (\text{left}+\text{right})/2 \) (middle index, between left and right)
  – Repeat
    • if \( v == A[m] \), we found \( v \), so we are done (break).
    • else if \( v < A[m] \), then \( \text{right} = m - 1 \).
    • else if \( v > A[m] \), then \( \text{left} = m + 1 \).

• We have reduced our search range in half, with a few instructions.
• For more flexibility \( \text{left} \) and \( \text{right} \) can be given as arguments to the function:
  \[
  \text{int search}(\text{int } A[], \text{int } N, \text{int } v, \text{int left, int right})
  \]

• See animation: [https://www.cs.usfca.edu/~galles/visualization/Search.html](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.

<table>
<thead>
<tr>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . . .</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>middle</th>
<th>Action (comparison)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Alice</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Cathy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Emma</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>John</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Paul</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Sam</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Tom</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

/* 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.)

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>middle</th>
<th>Action (comparison)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
<td>Dan &lt; John (go left)</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>Dan &gt; Cathy (go right)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>Dan &lt; Emma (go left)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td>Indexes cross, stop. Not found</td>
</tr>
</tbody>
</table>

/* 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.*/

```c
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.     else if (v < A[m])
8.       right = m-1;
9.     else
10.        left = m+1;
11.   }
12.   return -1;
13. }
```
/* 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.   {
6.     int m = (left+right)/2;
7.     if (v == A[m]) return m;
8.     if (v < A[m])
9.       right = m-1;
10.    else
11.       left = m+1;
12.   }
13.   return -1;
14. }

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 => \[ \lceil \lg N \rceil + 1 \]
   Verify with other cases, e.g. N:
   17, 31, 33

See Extra Material slides at the end for more details.
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!
Interpolated search
Money winning game:

- There is an array, A, with 100 items.
- The items are values in range [1, 1000].
- A is sorted.
- Values in A are hidden (you cannot see them).
- You will be given a value, val, to search for in the array and need to either find it (uncover it) or report that it is not there.
- You start with $5000. For a $500 charge, you can ask the game host to flip (uncover) an item of A at a specific index (chosen by you). You win whatever money you have left after you give the correct answer. You have one free flip.

<table>
<thead>
<tr>
<th>Value, val, you are searching for.</th>
<th>What index will you flip? (this is a TV show so indexes starts from 1, not 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>524</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>99</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>48</td>
</tr>
</tbody>
</table>
Money winning game – **Version 2 only specific indexes can be flipped.**

- There is an array, A, with 100 items.
- The items are values in range [1,100].
- A is sorted.
- Values in A are hidden (you cannot see them).

- You will be given a value, val, to search for in the array and need to either find it (uncover it) or report that it is not there.
- You start with $5000. For a $500 charge, you can ask the game host to flip (uncover) an item of A at a specific index (chosen by you). You win whatever money you have left after you give the correct answer. You have one free flip.

### Table: Index Selection

<table>
<thead>
<tr>
<th>Value, val, you are searching for.</th>
<th>What index will you flip?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1?, 10?, 25?, 50?, 75?, 90?, 100?</td>
</tr>
<tr>
<td>524</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>99</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>49</td>
<td>100</td>
</tr>
</tbody>
</table>
Interpolated Binary Search

- Similar to binary search, but I want an ‘educated guess’.

- E.g. given sorted array, A, of 100 numbers in range [0,1000], if you search in A for the value below, what index will you look at first?
  
  Assume it is a money-winning game and that for each trial/question, you loose some of the prize money. Which indexes would you pick?
  
  - 524 Look at index: 1?, 10?, 25?, 50?, 75?, 90?, 100?
  - 100 Look at index: 1?, 10?, 25?, 50?, 75?, 90?, 100?
  - 10 Look at index: 1?, 10?, 25?, 50?, 75?, 90?, 100?

**Direct formula for \([a,b) -> [s,t)\):**

\[ z = \frac{x-a}{b-a} (t-s) + s = (x-a) \frac{t-s}{b-a} + s \]

As a check, see that \(a->s\) and \(b->t\).

**Here:** value range \([M_n, M_x]\), index range \([s, t]\).

Use the \([M_n, M_x] -> [s, t]\) transformation and use for \(x\) the value you are searching for.

The result, \(z\), is the index, \(i\), you are looking for.
Range Transformations
(Math review)

• Draw and show the mappings of the interval edges.

• \([0,1) \rightarrow [0,n)\)
  \[y = xn\]

• \([a,b) \rightarrow [0,1) \rightarrow [0,n)\)
  \[y = \frac{x-a}{b-a}, \quad z = yn\]

• \([a,b) \rightarrow [0,1) \rightarrow [s,t)\)
  \[z = y(t-s) + s\]

if \([a,b] \rightarrow [0,n)\):
  \[z = \frac{x-a}{b-a+1}n\]

As a check, see that \(a- \rightarrow 0\) and \(b- \rightarrow y < n\).

Direct formula for \([a,b) \rightarrow [s,t)\):
  \[z = \frac{x-a}{b-a}(t-s) + s\]

As a check, see that \(a- \rightarrow s\) and \(b- \rightarrow t\).
Extra Materials
(Not Required)
EXTRA material: Binary Search
Deriving and Verifying the Exact Formula

• Max new candidates at next iteration:
  – \( \left\lceil \frac{N-1}{2} \right\rceil \)
  – \((N-1)\) b.c. middle position is removed
  – Rounded up because I pick the larger one of the two halves.

• Examples verifying the formula: \( \left\lceil \lg N \right\rceil + 1 \)

\[
\begin{array}{l}
N (== 2^5) \\
32 = 2^5 \\
16 = 2^4 \\
8 = 2^3 \\
4 = 2^2 \\
2 = 2^1 \\
1 = 2^0 \\
0 – stop, not counted
\end{array}
\]

5+1 = 6 iterations

\[
\begin{array}{l}
N (< 2^5): \\
17 \\
8 \\
4 \\
2 \\
1 \\
0 – stop \\
5 iter = 4 + 1
\end{array}
\]

\[
\begin{array}{l}
N (< 2^5): \\
31 \\
15 \\
7 \\
3 \\
1 \\
0 – stop \\
5 iter = 4 + 1
\end{array}
\]

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

When \( N \) is NOT an exact power, e.g. \( 2^{p-1} < N < 2^p \), then we have \( p \) iterations.
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.
Insertion sort code (Sedgewick)

```c
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++)
        complexch(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;
    }
}
```