1. **Searching**: finding and receiving a specified item from a large dictionary of similar/dissimilar items.
   1. Also applicable to the problem space
      1. Artificial intelligence
      2. Finding way out of maze
   2. Assumptions
      1. Each item has a *key* that is its handle.
      2. Keys are unique.
         1. Not always true.
      3. Search stops when an item is found.
         1. Not realistic when there are multiple ones.
      4. An unsuccessful search is always the worst case.
2. Searches
   1. Types
      1. **Internal search**: all items in electronic memory.
         1. Main focus
      2. **External search**: some items in external memory.
         1. Disk
         2. Implies some parts of external contents must be brought into memory and replaced with others continuously.
      3. **Off-line search**: sequence of search requests can be examined and manipulated before being processed.
      4. **On-line search**: requests are processed one by one in real time.
         1. Results on one requests returned before the next one is considered.
            1. The more difficult of the 2
3. **Dictionaries**: (assumed) contiguous list (array-based) of items.
   1. Types
      1. **Static dictionary**: can neither add nor remove entries.
         1. Printed phone book
      2. **Dynamic dictionary**: can add or remove entries.
         1. Club membership
      3. **Semi-dynamic dictionary**: can add but not remove entries.
         1. Obituaries
   2. Representation
      1. Referred to as the\_list.
      2. Contain records and keys
         1. Records: sought data.
         2. Keys: label of record sought.
4. Data Structures to Search
   1. Keys compared for equality or relative ordering through operators and strcmp().
   2. Records contain a key member, which is the primary means of identifying that record.
      1. Record can nonetheless be found through several keys.
5. Search Function
   1. After successful execution, retrieve() will assign the value of the index of each cell being visited to integer variable pos.
   2. If key member of record retrieved is the one sought, it returns success.
      1. pos integer will contain the position of the record found.
6. **Sequential search**: searches the dictionary sequentially, item by item, comparing item’s key to *target* key until it finds it or returns a value indicating it was unsuccessful.
   1. Simplest type of search algorithm
   2. Only way to search through an unordered list.
   3. Analysis
      1. Key comparison is the most expensive operation, especially with string keys.
      2. Assumptions about the list
         1. *n* elements
         2. comparisons take time *tc*
      3. Cases
         1. Best-case scenario: found on first try
            1. Computation time = *tc*
         2. Worst-case scenario: search unsuccessful
            1. Computation time = *ntc*
         3. Average: assuming equal probability.
            1. Computation time
      4. Given that *tc* is constant, efficiency ~ *n*.
   4. Types of Algorithms
      1. **Intractable algorithms**: can only work with low values of *n*.
      2. **Polynomial algorithms**
      3. **Linear complexity algorithm**
7. **Binary search**: bisects the list to limit search space.
   1. Most popular and efficient of all search algorithms.
      1. If dictionary can be ordered prior to search, there is a significant improvement in efficiency.
         1. When comparing keys, it can easily be determined which side of the list contains the key.
         2. Pruning the list reduces complexity.
   2. Unordered lists
      1. Hard to maintain
      2. Searches are limited to sequential searches
   3. Ordered list: contains records in a sort of order (numerical, alphabetical).
      1. Much more efficient
         1. Can be *pruned* to limit search space.
      2. Keeping list ordered when adding/deleting items can be costly.
   4. Contiguous lists > linked lists for binary searches
      1. Easy to bisect
      2. Easy to maintain definition parts of list
   5. Simple but not trivial to implement in program.
      1. Search begins at midpoint of list.
         1. Easy to find in array
         2. Not easy in linked list
      2. Compares key at midpoint element to target key.
         1. If record at midpoint matches target
            1. Success
         2. Otherwise
            1. Continue bisecting active part of list until reduced to only one record.
            2. If not sought target

Failure

* + 1. Process
       1. Set
          1. Top
          2. Middle
          3. Bottom
       2. Bisect
       3. Repeat until key is found.
  1. Analysis
     1. Through a non-trivial derivation
     2. Efficiencies
        1. Best case = *tc*
        2. Worst case = *tc* × log2(*n*)
        3. Average case = log2(*n*) × 0.5*tc*

1. **Sorting**: converting an unordered list into an ordered list according to some convention.
   1. Conventions
      1. Alphabetically
      2. Reverse alphabetical
      3. Increasing numerical order
      4. Decreasing numerical order
   2. Types of Algorithms
      1. **Simple algorithms**: inefficient (*n*2) but easy to write.
         1. **Insertion sort**: compares the value of one cell to its adjacent element, and swaps it if out of place.
            1. Analogous to how people normally sort things.
            2. Divides memory into 2 regions

**Sorted region**

Initially the **head**.

**Unsorted region**

Initially the entire list.

Upon completion, the entire list.

* + - * 1. Steps

Pick 1 item at a time.

Insert item where it belongs in relation to others around it.

* + - * 1. There is only one memory element, in which the entire algorithm must occur.

List

Array

* + - * 1. Algorithm

2 embedded loops

Outer loop: moves sorted region by 1 every iteration; shrink unsorted region by 1 every iteration.

Inner loop: compares every item with its adjacent cell on its left, until no more swapping is necessary (i.e., can run multiple times).

Until unsorted region is empty, do…

Take first element in unsorted region

Until unsorted element >= sorted element, do…

Compare with left adjacent element in sorted region

If unsorted element < sorted element, swap.

Else, continue

Stop

Return

* + - 1. **Selection sort**: selects the next smallest (or largest) list element and directly places it in the appropriate location.
      2. **Bubble sort**: inserted element “bubbles” up to its proper location in the sorted part of the list, left to right.
    1. **Intermediate algorithms**: somewhat of both, but less of each than the 2 other categories.
       1. **Shell sort**: almost completely sorts, then use insert sort.
    2. **Complex algorithms**: highly efficient (*n*log2*n*) but difficult to code.
       1. **Mergesort**
          1. Uses recursion
       2. **Quicksort**
    3. **Tree-based algorithms**: easy to visualize, intuitive, and efficient; not difficult to write, as they use recursion and certain data types.
       1. **Tree sort**
       2. **Heap sort**