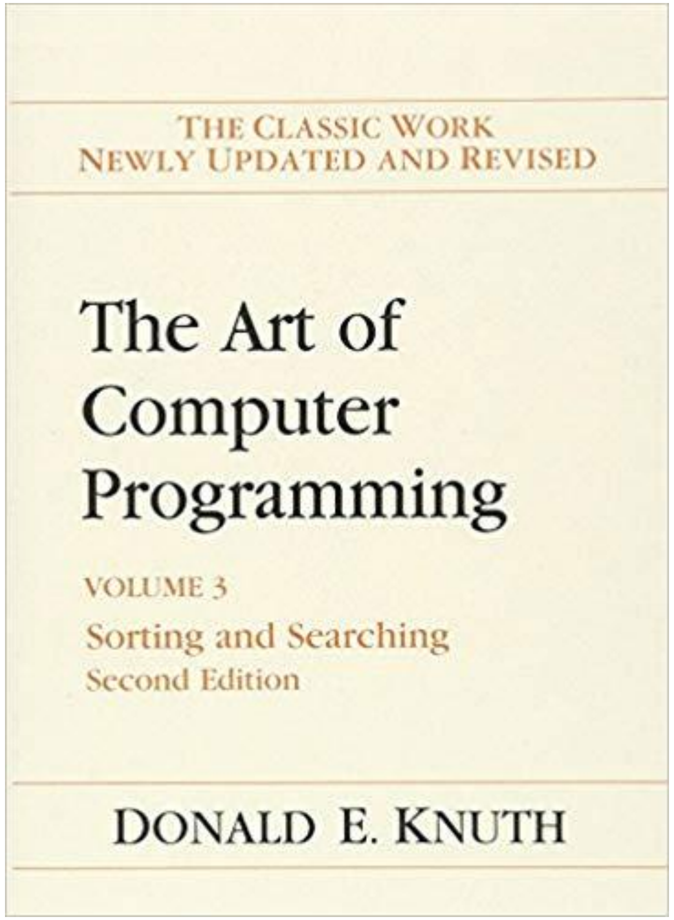
Some Simple Algorithms and Data Structures

Chapter 12

Donald E Knuth



Keys to efficiency

- Modern computers are fast
  - So sometimes doing things the hard way is okay
- But not that fast
  - So, we need to be wise
  - Choose efficient algorithms
  - Not clever coding tricks
  - A “wise man” once said “you read code more than you write it”
- Learn from the array of past work to your benefit
  - Develop an understanding of to complexity of a problem
  - Think about how to decompose it
  - Relate those sub-problems to existing solutions

Search Algorithms

- "A **search algorithm** is a method for finding an item or group of items with specific properties within a collection of items. We refer to the collection of items as a **search space**." (p. 234)
- Search space can be
  - A fixed collection such as a list, string, tuple or dictionary
  - A more abstract collection like the set of all integers

Linear Search

- How does Python search a list?

```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
    return False
```

- Is this efficient?
  - We don’t know if the list is sorted
  - Will check every element until found
  - O(len(L)) – “at best”
    - Assumes fetching elements is a constant step
    - Remember a list can contain multiple types of elements

Indirection

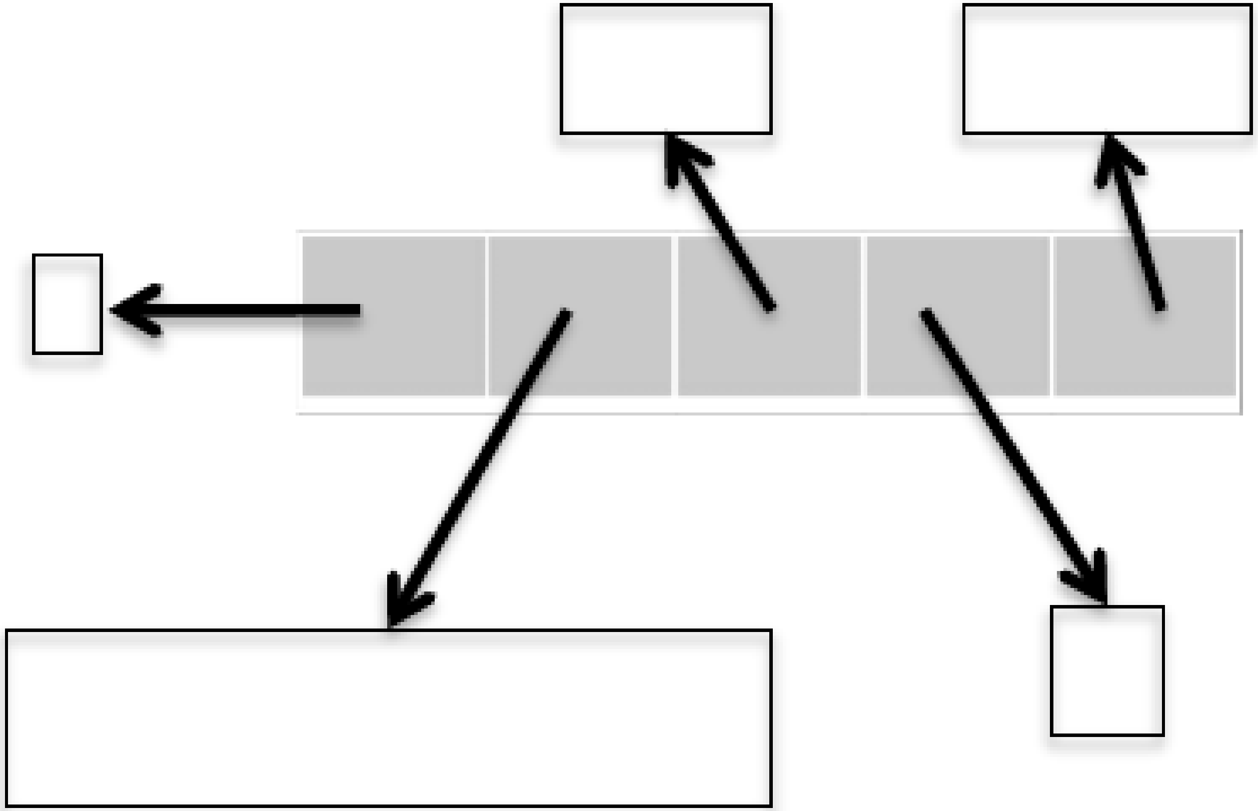


Figure 10.1 Implementing lists

Searching a sorted list

```
# Linear Search on a sorted list
def search(L, e):
    """Assumes L is a list, the elements of which are in
    ascending order.
    Returns True if e is in L and False otherwise"""
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
    return False
```

Bisection Search

- We have already done something like this
  - Chapter 3 – binary search for square root
- 4 simple steps
  1. Pick an index, `i`, that divides the list `L` roughly in half.
  2. Ask if `L[i] == e`.
  3. If not, ask whether `L[i]` is larger or smaller than `e`.
  4. Depending upon the answer, search either the left or right half of `L` for `e`.
- Step 4 looks recursive to me
- Let's look at an implementation

```
def search(L, e):
    """Assumes L is a list, the elements of which are
in
          ascending order.
       Returns True if e is in L and False otherwise"""

    def bin_search(L, e, low, high):
        #Decrements high - low
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid: #nothing left to search
                return False
            else:
                return bin_search(L, e, low, mid - 1)
        else:
            return bin_search(L, e, mid + 1, high)

    if len(L) == 0:
        return False
    else:
        return bin_search(L, e, 0, len(L) - 1)
```

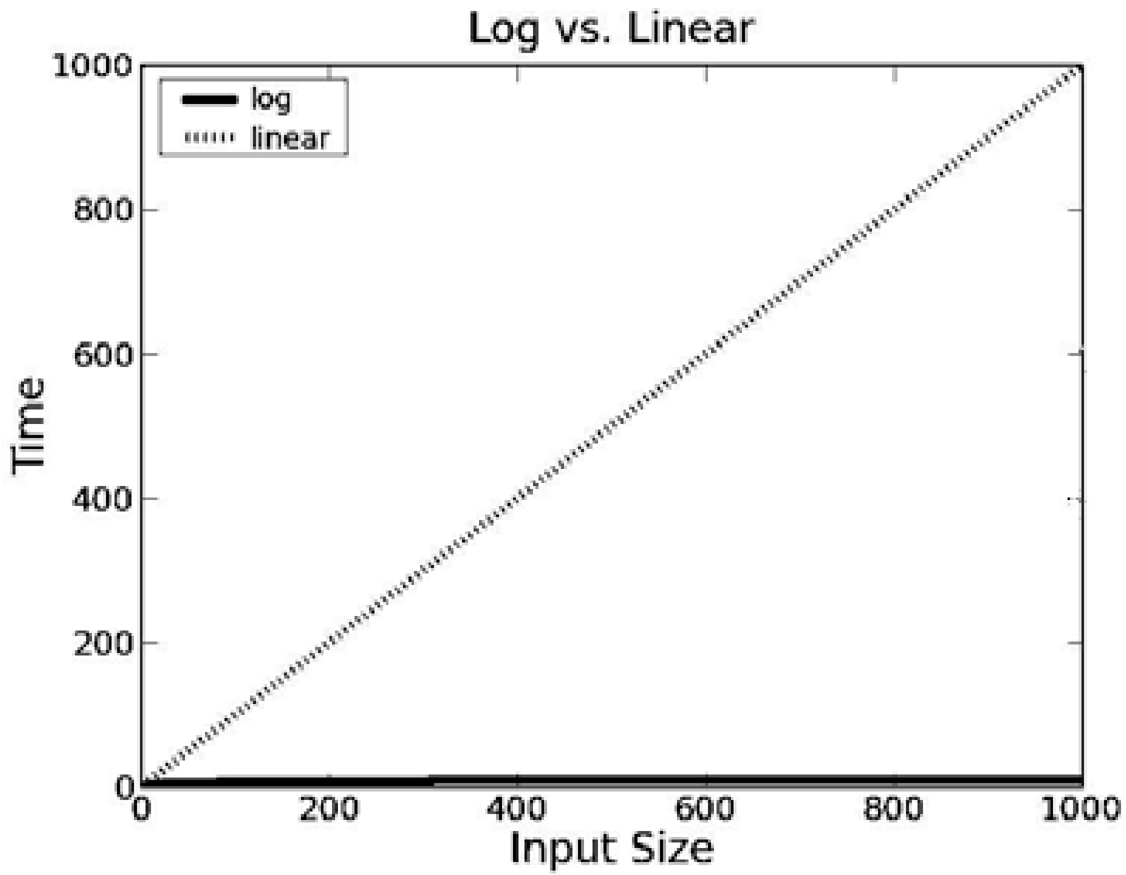
Binary Search implementation

- `search()` is a **wrapper function**
  - Provides the interface to the client
  - Hides the implementation in `bin_search()`
- What is the complexity
  - Depends on the number of levels of recursion
  - When does it stop?
    - `e` found OR `low == high`
    - So `high-low` is the decrementing function
  - O(log(len(L)))

Binary Search lingering questions

- Why does the code use `mid+1` rather than `mid` in the second recursive call?
- Why use the `bin_search()` with the `low` and `high` parameters rather than just using slices of L?

O(len(L)) $\stackrel{?}{=}$ O(log(len(L)))



A dialog

- Q: Should I always sort a list before searching so I can use a binary sort?
- A: Is it faster to sort a list and then do that binary search – than to just do a linear search?
  - If O_S(L) = the complexity of the search
  - Is O_S (L) + O(log(len(L))) $<$ O(len(L))
- Q: I dunno, is it?
- A: Sadly, no – a sort algorithm has to look at every element in a list
- Q: So why did we just learn about binary search?
- A: Because, Grasshopper, it depends

It depends?

- Are we going to search more frequently than we add elements to the list or alter elements in the list?
  - If k = the number of times we expect to sort the list
  - Is O_S (len(L)) + k*O(log(len(L))) < k*O(len(L))
  - It still depends on O_S
- Do deletes matter?

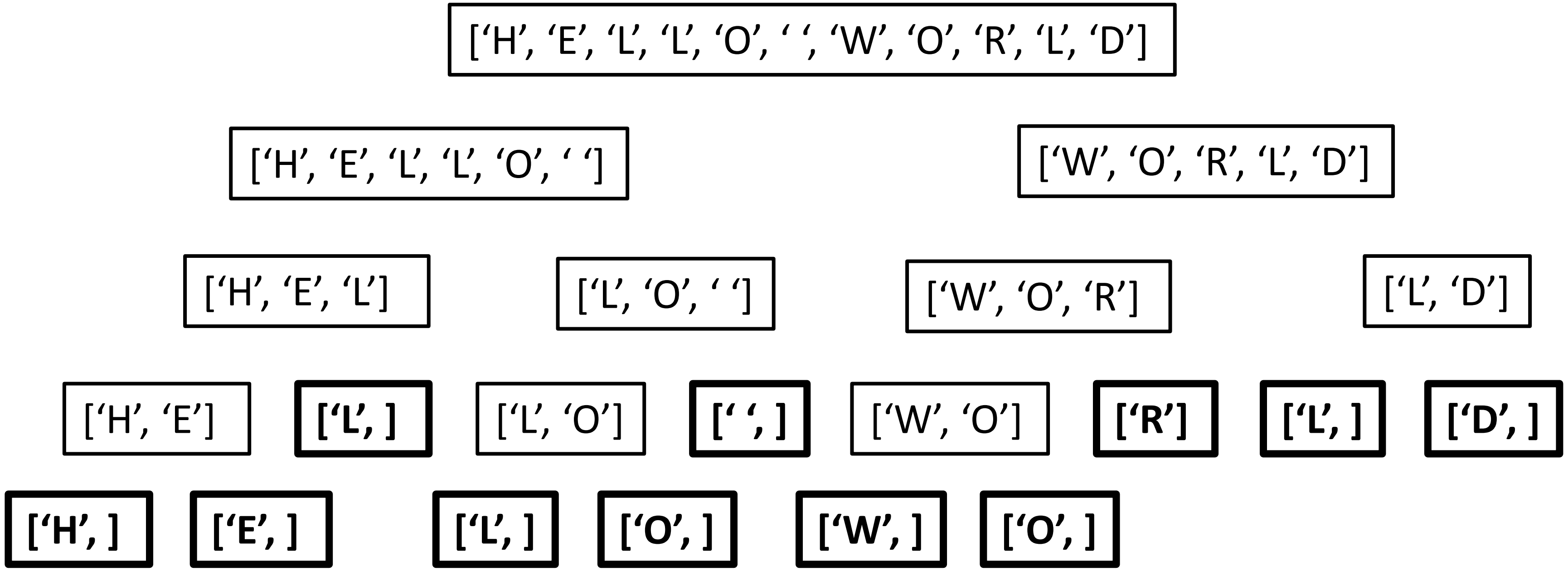
Selection sort

- Simple but **inefficient** sorting algorithm
- **Loop invariant**
  - Condition that is true at the beginning and end of each loop
  - List contains two partitions
    - L[0:i] (prefix) and L[i+1:len(L)]
    - Prefix is sorted and all elements are less then any elements in suffix
- Base case – i = 0 so prefix is empty, suffix is full list
- Each iteration (Induction) – move the smallest element in suffix to last position of prefix
- Termination – prefix is full list sorted, suffix is null
- $\theta(len(L)^2)$

Merge Sort

- An example of a **divide-and-conquer** algorithm
  - Continue to divide until a threshold (minimum) problem size is reached
  - A size and number of sub problems
  - An algorithm to combine the results of the sub solutions
- Applying divide-and-conquer to sorting
  - A list with 0 or 1 elements is intrinsically sorted
  - If a list has more than one element – split and merge sort each half
  - Merge the resulting
  - Merging can take advantage of the fact the partitions are already sorted

Merge Sort – first divide



Merge Sort – now conquer

['H',] ['E',] ['L',] ['O',] ['W',] ['O',]

['E', 'H'] ['L',] ['L', 'O'] [' ',] ['O', 'W'] ['R'] ['L',] ['D',]

['E', 'H', 'L'] [' ', 'L', 'O''] ['O', 'R', 'W'] ['D', 'L']

[' ', 'E', 'H', 'L', 'L', 'O' ['D', 'L', 'O', 'R', 'W']

[' ', 'D', 'E', 'H' 'L', 'L', 'L', 'O', 'O', 'R', 'W']

Determining Complexity

- What is the **Computational Complexity** of our implementation?
- We will divide and conquer this

Complexity of merge()

```
def merge(left, right, compare):
    result = []
    i,j = 0, 0
    while i < len(left) and j < len(right):
        if compare(left[i], right[j]):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result
```

- Looking at the inner – what is the complexity of `merge()`
  - How many times will we compare `a[0] < b[0]`
    - Length of the longer list
    - We'll still same len(L)
  - How many times will we copy an element to c
    - Once per element
    - We'll say len(L1) + len(L2)
  - So the whole thing is linear (O(len(L))

Complexity of merge_sort()

```
def merge_sort(L, compare = lambda x, y: x < y):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle], compare)
        right = merge_sort(L[middle:], compare)
        return merge(left, right, compare)
```

- How many times do we recurse into `merge_sort()`?
  - log(len(L))

Putting it all together

- merge() is called once per execution of merge_sort()
  - Overall complexity is O(merge_sort) * O(merge)
  - Complexity of merge_sort() is O(log(n))
  - Complexity of merge() is (O(len(L))
- Computational Complexity is O(n log(n))

What is the deal with `compare`?

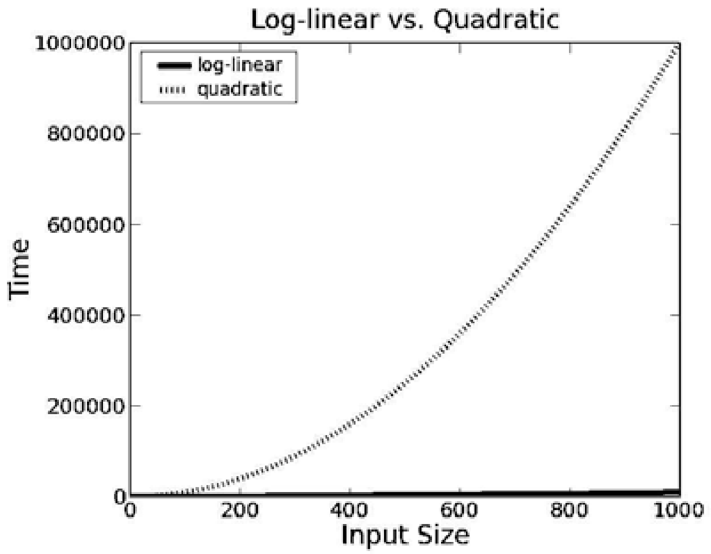
- Recall `compare = lambda x,y: x < y`
- Functions are ‘first class’ objects
- This allows us to drive the sort
    - Reverse order `lambda x,y: x > y`
    - Can define our own sort order
        - Names – last, first or first, last
        - Addresses – zip code, street name, street #

Sort algorithms

- **Exchange sorts** – Bubble, Cocktail Shaker, Odd-Even, Comb, Gnome, Quicksort, Slowsort, Stooge, Bogo
- **Selection sorts** – Selection, Heapsort, Smoothsort, Cartesian Tree, Tournament, Cycle, Weak-heap
- **Insertion sorts** – Insertion, Shellsort, Splaysort, Tree, Library, Patience Sorting
- **Merge sorts** – Merge, Cascade, Oscillating, Polyphase
- **Distribution sorts** – American Flag, Bead, Bucket, Burstsort, Counting, Pigeonhole, Proxmap, Radix, Flashsort
- **Concurrent sorts** – Bitonic, Batcher-odd-even-mergesort, Pairwise sorting network
- **Hybrid sorts** – Block merge, Timsort, Intosort, Spreadsort, Merge-Insertion
- **Other** – Topological sorting, Pancake sorting, Spaghetti

Comparing Selection and Merge Sorts

- Computational Complexity
  - Selection Sort ∈ $O(n^2)$
  - Merge Sort ∈ $O\big(n\ log(n)\big)$
- Space Complexity
  - Selection Sort is in-line so Constant
  - Merge Sort ∈ $O\big(len(L)\big)$



Sorting in Python

- L.sort()
  - sorts list L
- L2 = sorted(L1)
  - L2 gets a sorted copy of L1
  - L1 remains unchanged
  - Can be used on other collections

More sorting in Python

- Python uses **timsort** which is a hybrid algorithm
  - Assumes most cases a list is partially sorted
  - Worst case performance – same as **merge_sort** O(n log(n))
  - Named for Tim Peters who developed it.

- `sorted` method

```
Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in
    ascending order.
    A custom key function can be supplied to customize the sort
    order, and the reverse flag can be set to request the result in
    descending order.
```

Hash Tables

- Remember dictionaries (`dict`) from Chapter 5?
- The key is must be **hashable** converted to an integer
  - Needs to be a *reasonable* integer **hash value**
  - Hash values are used to index into a list
  - Access time is nearly constant
- A **hash function** takes data from a large problems space and converts them to indices within a small index space
  - Hash values may be **many-to-one** (i.e. non-unique)
  - This can cause **collisions**
  - A good hash function will have a **uniform distribution**

Int_Dict – A simple dictionary for integers

- The **dictionary** is implemented as a list of lists of tuples
  - Each **tuple** is a key/value pair
  - Each **inner list** represents a **collision set**
  - Each **outer list** represents a **bucket**
  - The number of **buckets** is defined when IntDict is instantiated
- Our **hash function** is the key ‘modulo’ (%) the number of available buckets
- Access depends on the ratio between number of buckets and size of the dictionary
  - And of course – the efficiency of the hash function

Other Hash Keys?

- Checksum
  - Exclusive Or (XOR) bytes or words
- Object.__hash__() method