Lecture 12

Fundamental Algorithms

Burkay Genç, Ahmet Selman Bozkır, and Selma Dilek
31/05/2023

PREVIOUS LECTURE

- Object Oriented Programming
  - A different way of thinking about programming
- Classes
- Objects

TODAY

- Searching
- Sorting

FUNDAMENTAL ALGORITHMS

Algorithmic Complexity

- A problem can be solved with many different algorithms
- Some will take seconds, others will take years
- Algorithm design is an important part of engineering
- **Complexity** shows how an algorithm will perform on **very large inputs**

Fundamental Algorithms

- There are two types of algorithms that are frequently used by other algorithms
  - *Searching* algorithms : search within a list for an item
  - *Sorting* algorithms : sort a list of items in ascending order
- General **algorithm design methodology**:
  - Develop an understanding of the **complexity** of the problem
  - Think about how to **break the problem into subproblems**
  - Solve subproblems using **existing efficient algorithms**

SEARCHING

Search Algorithm

- 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**
- Many problems in real life can be reduced to search problems

Specification

```
def lin_search(L, e):
  """Assumes L is a list.
  Returns True if e is in L and False otherwise"""
```

- How is that different than `e in L`?

Linear Search

```python
def lin_search(L, e):
  """Assumes L is a list.
  Returns True if e is in L and False otherwise"""
  for i in range(len(L)):
    if L[i] == e:
      return True
  return False
```

- This is how Python implements search
- This takes **linear** time to find an item
  - We need to look at each item once to find a specific item
  - If we are **lucky**, it can be the **first** item in the list
    - The for loop runs only once
  - If we are **unlucky**, it can be the **last** item in the list
    - The for loop runs `len(L)` times
- **Worst case** is the for loop runs `len(L)` times
  - `len(L)` -> Size of input -> linear time algorithm

Binary Search

- If we know nothing about the list and the items, then linear time search is the best we can do
- But consider searching for a **word in the dictionary**
  - Do you go over each word one by one to find the word?
  - You actually jump to a page
    - Check the words on that page,
      - Keep jumping into the half that makes sense
  - You can do that because a dictionary is **sorted**

Binary Search

- If you are given a list of items in sorted order, can you search for a specific item faster than linear time?
  - The answer is YES
- strategy
  - Look at the item in the middle
  - If it is equal to the item you are searching for,
    - return True
  - Else if it is less than what you are searching for
    - Repeat the search with the right half
  - Else,
    - Repeat the search with the left half

Example

- example
  - Searching for 7
  - Given a sorted list:

```
li = [1,3,4,5,7,12,17,18,19,24,32,33,35,40]
```

- Is 7 in this list?

Example

- There are

```
len(li)
```

```
## 14
```

items in the list

- Check the middle one

```
li[len(li) // 2]
```

```
## 18
```

- 18 is **greater** than 7, so we repeat with the **left** half.

Example

```
li = li[0:(len(li)//2)]
li
```

```
## [1, 3, 4, 5, 7, 12, 17]
```

- Check the middle one

```
li[len(li) // 2]
```

```
## 5
```

- 5 is **less** than 7, so we repeat with the **right** half.

Example

```
li = li[(len(li)//2+1):len(li)]
li
```

```
## [7, 12, 17]
```

- Check the middle one

```
li[len(li) // 2]
```

```
## 12
```

- 12 is **greater** than 7, so we repeat with the **left** half.

Example

```
li = li[0:(len(li)//2)]
li
```

```
## [7]
```

- Check the middle one

```
li[len(li) // 2]
```

```
## 7
```

- We found 7 in the list!

Implementation

- Let's turn this into python code

```
def bsearch(L, e, start, end):  # Search e in L[start:end]
  if start == end:
    return L[start] == e:
  else:
```

Implementation

- Now, check the middle item:

```
def bsearch(L, e, start, end):  # Search e in L[start:end]
  if start == end:
    return L[start] == e:
  else:
    middle = (start+end)//2     # middle item of the list
    if L[middle] == e:
      return True
    ...
```

Implementation

- Now, recurse into the correct half:

```
def bsearch(L, e, start, end):  # Search e in L[start:end]
  if start == end:
    return L[start] == e
  else:
    middle = (start+end)//2     # middle item of the list
    if L[middle] == e:
      return True
    elif e < L[middle]:
      return bsearch(L, e, start, middle)  # keep searching in the left half
    else:
      return bsearch(L, e, middle + 1, end) # keep searching in the right half
```

Testing

- Let's test

```
L = [1,3,4,5,7,12,17,18,19,24,32,33,35,40]
bsearch(L, 7, 0, len(L)-1)
```

```
## True
```

```
bsearch(L, 1, 0, len(L)-1)
```

```
## True
```

```
bsearch(L, 40, 0, len(L)-1)
```

```
## True
```

```
bsearch(L, 0, 0, len(L)-1)
```

```
## False
```

```
bsearch(L, 50, 0, len(L)-1)
```

```
## False
```

```
bsearch(L, 9, 0, len(L)-1)
```

```
## False
```

Improvement

- It is not pretty to write `bsearch(L, 7, 0, len(L) - 1)`
  - Too many arguments to call

```
def bin_search(li, it):
  return bsearch(li, it, 0, len(L) - 1)

bin_search(L, 7)
```

```
## True
```

```
bin_search(L, 45)
```

```
## False
```

Justification

- Is `bin_search` really faster than `lin_search`?
  - If the list is sorted, on the average, YES

```
import time
from random import gauss
li = [gauss(0,1) for i in range(1000000)] # Create a list of one million random numbers
li.sort()                                 # sort the list
start = time.process_time()               # mark the start of lin_search
for i in range(1, 20):                    # search for 20 different numbers
  res = lin_search(li, li[50000*i])
lin_elapsed = time.process_time() - start # mark the end
start = time.process_time()               # do the same for bin_search
for i in range(1, 20):
  res = bin_search(li, li[50000*i])
bin_elapsed = time.process_time() - start
print("Time spent in linear search:", lin_elapsed)  # print the results
```

```
## Time spent in linear search: 1.25
```

```
print("Time spent in binary search:", bin_elapsed)
```

```
## Time spent in binary search: 0.0
```

Justification

- What really happened?
  - At each recursive call, binary search gets rid of half of the current list
  - In the first call it gets rid of 500000 items,
  - Then 250000 items,
  - Then 125000 items,
  - ...
- How many times can you do that?
  - When you hit 1 item, you have to stop
- This is called **logarithms** in base two
  - Recursion will run $log_2(len(li))$ times
  - $log_2(1000000) << 1000000$
  - log time << linear time

SORTING

Sorting

- A sorted list is easier to search
- But how can we sort a list **as fast as possible**?
- Python's built-in sort function is very efficient
  - It runs in $O(n \log n)$ time
    - That is a special notation computer scientists use to represent speed of algorithms
    - $O(n^2) > O(n \log n) > O(n) > O(\log n) > O(1)$
    - binary search: $O(\log n)$
    - linear search: $O(n)$
    - python's sort: $O(n \log n)$

Selection Sort

- Python's sort is **fast**, use it!
- We provide **selection sort** only for practice purposes
- **strategy**
  - *loop invariant*
    - `L = prefix + suffix`
    - `prefix = L[0:i]`
    - `suffix = L[i:len(L)]`
    - at the end of i^{th} iteration
      - prefix is sorted
      - all items in suffix are greater than all items in prefix

Selection Sort

```
def selSort(L):
  """Assumes that L is a list of elements that can be compared using >.
  Sorts L in ascending order"""
  suffixStart = 0
  while suffixStart != len(L):
    #look at each element in suffix
    for i in range(suffixStart, len(L)):
      if L[i] < L[suffixStart]:
        #swap position of elements
        L[suffixStart], L[i] = L[i], L[suffixStart]
    suffixStart += 1

li = [9,8,7,6,5,4,3,2,1]
selSort(li)
li
```

```
## [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Complexity

- What is the complexity of selection sort?
  - the for loop runs $O(n)$ times
  - the while loop runs $O(n)$ times
  - overall: $O(n) * O(n) = O(n^2)$
- It is a very slow algorithm for large inputs

```
li = [i for i in range(10000, 1, -1)]
li2 = li.copy()
start = time.process_time()
selSort(li)
elapsed = time.process_time() - start
print(elapsed)
```

```
## 4.515625
```

```
start = time.process_time()
li2.sort()
elapsed = time.process_time() - start
print(elapsed)
```

```
## 0.0
```

How To Sort Fast?

- Fast sorting algorithms use an approach called **Divide and Conquer**
  - Divide the problem into two halves
  - Solve the problem on each half
  - Combine/merge the halves
- Examples
  - **mergesort**
  - **quicksort**

END OF THE COURSE