CS311 Data Structures
Lecture 01 — Introduction

Jyh-Ming Lien

June 3, 2018

(Abstract) Data Structures?

- What are they?
- Why do you have to learn data structures?
- Where will it be used (e.g. in CS 483)?

How to be a good computer engineer?

- Good engineers are lazy, otherwise
  - every door in a building
  - every light switch
  - every power outlet
  - every screw
  - ... would be different
- Lazy engineers spent minimum effort to solve a problem
  - never reinvent the wheel
  - never start from scratch
  - always *reuse* (but don't steal) existing tools.
- Lazy computer engineers write minimum code to solve a problem
- However, in CS 310, we start our code from scratch so we can learn
- Today's topic: How to become a lazy computer engineer?
  - Lazy computer engineers use **generics**
  - Lazy computer engineers use **recursion**
  - Lazy computer theoreticians use **asymptotic notation**

Generic Linked List

- What is a list (of integers)?
- Why do we need a linked list?
- What are the functions that we normally need to manipulate a list?
- Given an object x, how do we check if x is in the list? (we call this function, "find(x)")

Generic Linked List

- Now, what do I do if I need a list of strings? Do I need to re-design the whole list?
- But I am lazy, so what should I do?
- Approach 1:
- Approach 2:

Find Max

```java
class FindMaxDemo
{
    /**
     * Return max item in arr.
     * Precondition: arr.length > 0
     */
    public static Comparable findMax( Comparable [ ] arr )
    {
        int maxIndex = 0;

        for( int i = 1; i < arr.length; i++ )
            if( arr[ i ].compareTo( arr[ maxIndex ] ) > 0 )
                maxIndex = i;

        return arr[ maxIndex ];
    }

    /**
     * Test findMax on Shape and String objects.
     */
    public static void main( String [ ] args )
    {
        Shape [ ] sh1 = { new Circle( 2.0 ),
                          new Square( 3.0 ),
                          new Rectangle( 3.0, 4.0 ) };

        String [ ] st1 = { "Joe", "Bob", "Bill", "Zeke" };

        System.out.println( findMax( sh1 ) );
        System.out.println( findMax( st1 ) );
    }
}
```

Find Max

```java
    // Generic findMax, with a function object.
    // Precondition: a.size( ) > 0.
    public static <AnyType>
    AnyType findMax( AnyType [ ] arr, Comparator<? super AnyType> cmp )
    {
        int maxIndex = 0;

        for( int i = 1; i < arr.size( ); i++ )
            if( cmp.compareTo( arr[ i ], arr[ maxIndex ] ) > 0 )
                maxIndex = i;

        return arr[ maxIndex ];
    }

class CaseInsensitiveCompare implements Comparator<String>
{
    public int compare( String lhs, String rhs )
      { return lhs.compareToIgnoreCase( rhs ); }
}

class TestProgram
{
    public static void main( String [ ] args )
    {
        String [ ] arr = { "ZEBRA", "alligator", "crocodile" };
        System.out.println( findMax( arr, new CaseInsensitiveCompare( ) ) )
    }
}
```

Recursion

- Fibonacci numbers $\mathrm{fib}(n)$:

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{if } n > 1 \end{cases} \tag{1}$$

- Example: The first 10 Fibonacci numbers are:
$\{0, 1, \_\_\_\_, \_\_\_\_, \_\_\_\_, \_\_\_\_, \_\_\_\_, \_\_\_\_, \_\_\_\_, \_\_\_\_\}$

Our First Algorithm

- Problem: What is $\mathrm{fib}(200)$? What about $\mathrm{fib}(n)$, where n is any positive integer?

Algorithm 3.1: $\mathrm{fib}(n)$

if $n = 0$
  then return (0)
if $n = 1$
  then return (1)
return $(\mathrm{fib}(n-1) + \mathrm{fib}(n-2))$

- Questions that we should ask ourselves.
  1. Is the algorithm correct?
  2. What is the running time of our algorithm?
  3. Can we do better?

Analyze Our First Algorithm

- Is the algorithm correct?
  - Yes, we simply follow the definition of Fibonacci numbers
- How fast is the algorithm?
  - If we let the run time of $\text{fib}(n)$ be $T(n)$, then we can formulate

$$T(n) = T(n-1) + T(n-2) + 3 \approx 1.6^n$$

  - $T(200) \geq 2^{139}$
  - The world fastest computer , which can run 2^{56} instructions per second (93 Peta FLOPS, Peta$=10^15$) , will take 2^{83} seconds to compute. (2^{83} seconds $= 3 \times 10^8$ billion years, Sun turns into a red giant star in 4 to 5 billion years, the Universe is about 13.82 billion years old)
  - Can Moose's law, which predicts that CPU get 1.6 times faster each year, solve our problem?
  - No, because the time needed to compute $\text{fib}(n)$ also have the same "growth" rate
    - if we can compute $\text{fib}(100)$ in exactly a year,
    - then in the next year, we will still spend a year to compute $\text{fib}(101)$
    - if we want to compute $\text{fib}(200)$ within a year, we need to wait for 100 years.

Improve Our First Algorithm

- Can we do better?
- Yes, because many computations in the previous algorithm are repeated.

Algorithm 3.2: $\text{fib}(n)$

comment: Initially we create an array $A[0 \cdots n]$

$A[0] \leftarrow 0, A[1] \leftarrow 1$
for $i = \{2 \cdots n\}$
  do $A[i] = A[i-1] + A[i-2]$
return $(A[n])$

Theoretical analysis of time efficiency

- Provide *machine independent* measurements
- Estimate the bottleneck of the algorithm
- The size of the input increases $\rightarrow$ algorithms run longer $\Rightarrow$. Typically we are interested in how efficiency scales w.r.t. input size
- To measure the running time, we could
  1. count all operations executed.
  2. or determine the number of the **basic operation** as a function of **input size**
- **Basic operation**: the operation that contributes most towards the running time

Orders of Growth

- Some of the commonly seen functions representing the number of the basic operation $C(n) =$
  1. n
  2. n^2
  3. n^3
  4. $\log_{10}(n)$
  5. $n \log_{10}(n)$
  6. $\log_{10}^2(n)$
  7. $\sqrt{n}$
  8. 2^n
  9. $n!$
- Can you order them by their growth rate?

Orders of Growth

- Test functions using some values

n	n^2	n^3	2^n	$n!$
10	10^2	10^3	1024	3.6×10^6
100	10^4	10^6	1.3×10^{30}	9.3×10^{157}
1000	10^6	10^9	1.1×10^{301}	
10000	10^8	10^12		

n	$\log_{10}(n)$	$n \log_{10}(n)$	$\log_{10}^2(n)$	$\sqrt{n}$
10	1	10	1	3.16
100	2	200	4	10
1000	3	3000	9	31.6
10000	4	40000	16	100

(see Weiss pg 203)

- Now, we can order the functions by their growth rate
$\log_{10}(n) < \log_{10}^2(n) < \sqrt{n} < n < n \log_{10}(n) < n^2 < n^3 < 2^n < n!$

Example: Maximum contiguous subsequence sum

Don't play: 0 gain

How Would you find Best Increase?

i	price	delta
1	886	0
2	890	4
3	880	-10
4	890	10
5	899	9
6	911	12
7	903	-8
8	913	10
9	920	7
10	924	4
11	927	3
12	921	-6
13	919	-2
14	887	-32
15	902	15

How is `payoff` computed for `start=5` and `end=12`?
For `start=7` and `end=10`?

Several names for the Problem

- Maximum contiguous subsequence sum (textbook)
- Maximum Subarray (wikipedia)
- Find `start` and `end` time with largest `payoff` out of all possible

Find a Solution

- Input is the array `delta[]`
- Output: `(start, end, payoff)` such that payoff is as large as possible
- Can optionally *not invest* for no payoff; return (-1,-1,0)

Algorithm 1: Brute Force

```
maxSubsequenceCube(int A[])
{
  bestPayoff = 0
  bestStart = -1
  bestEnd = -1
  for start=0 to A.length-1 {
    for end=start to A.length-1 {
      currentPayoff = 0
      for i=start to end {
        currentPayoff += A[i]
      }
      if(currentPayoff > bestPayoff){
        bestPayoff = currentPayoff
        bestStart = start
        bestEnd = end
      }
    }
  }
  return bestPayoff, bestStart, bestEnd
}
```

- `A[]` contains deltas
- Try every possible `start` and `end` (outer loops)
- Calculate increase from `start` to `end`
- Track the best seen
- Complexity?
- Anything better

Algorithm 2 Alternative: Convert to global Prices

```
maxSubsequenceQuad(int A[]){
  B = new array size A.length
  B[0] = A[0]
  for i=1 to B.length-1
    B[i] = B[i-1] + A[i]

  best = 0
  bestStart = -1
  bestEnd = -1
  for start=0 to A.length-1 {
    for end=start to A.length-1 {
      current = B[end] - B[start]
      if(current > best){
        best = current
        bestStart = start
        bestEnd = end
      }
    }
  }
  return best, bestStart, bestEnd
}
```

- Initially convert deltas in `A` to global prices in `B`
- First price doesn't matter as interested in changes
- Try every start and end
- Easy to calculate `currentPayoff`
- Memory overhead?

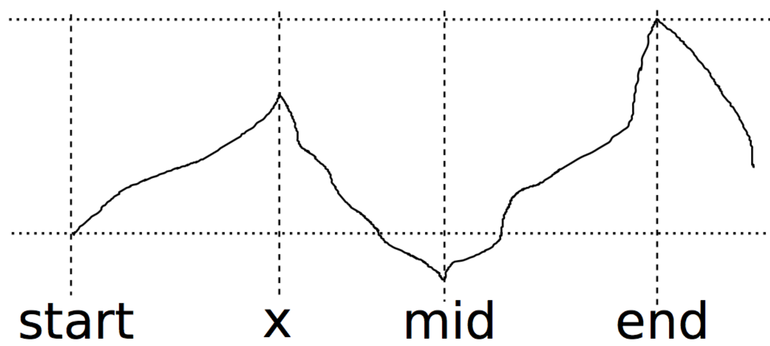
A Helpful Property

Proposition: The shortest maximum subsequence beginning at `start` and finishing at `end` contains no point `mid` between them with a lower value than `start`.

Proof by Contradiction:

- Suppose shortest max subsequence exists, looks like picture.
- `x` must be lower than `end`, o/w could form a shorter maximum subsequence `start` to `x`
- But if `mid` is lower then `start`, sequence `mid` to `end` has a larger increase than `start` to `end`.

Contradiction □



Consequence: If `mid` drops below `start`, reset `start` to `mid`
Create a faster algorithm based on this property.

Algorithm 3: Scan

```
maxSubsequenceLinear(int A[]){
  best = 0
  current = 0
  bestStart = -1
  bestEnd = -1
  start = 0
  for end=0 to A.length-1 {
    current += A[end]
    if(current > best){
      best = current
      bestStart = start
      bestEnd = end
    }
    else if(current < 0){
      start = end+1;
      current = 0;
    }
  }
  return best,bestStart,bestEnd;
}
```

- `A[]` contains deltas
- When sum `current` falls below zero, move `start` to `end` and reset
- Single pass over entire array

Max Subsequence Algorithms Synopsis

Comparisons

Given that array A has n elements,

- `maxSubsequenceCube()`: triply nested loops over entire array, $O(n^3)$
- `maxSubsequenceQuad()`: doubly nested loops over entire array, $O(n^2)$
- `maxSubsequenceLinear()`: single loop over entire array, $O(n)$

Intuition: for large arrays, `maxSubsequenceLinear()` will produce answers faster

Conclusion

- Lazy computer engineers do **generics**
- Lazy computer engineers do **recursion** (with care!)
- Lazy computer theoreticians do **asymptotic notation**

- It is not easy to be lazy; you need to try very hard!
- Read: Chapter 5
- Next week: More Big-O, List, Stacks, Queues