Algorithms and Time-Complexity

Dr Jonas Lundberg

Jonas.Lundberg@lnu.se

February 18, 2020

General

- An algorithm is:
  - *An unambiguous description of how to solve a specific problem.*
- The first ($\approx$ 300BC) non-trivial algorithm is *Euclidean Algorithm*.
  - Finds the greatest common divisor for two positive integers.
- The Persian mathematician Mohammed al-Khowârizmî was around year 800 the first to write down step-by-step algorithms for addition, subtraction, multiplication and division.
  - In Latin, his name became *Algorismus* and it is from this we have the word *algorithm*.

Problem: Wash your hair

- Q: How do you wash your hair?
- A: Start by wetting the hair. Then rub in shampoo and rinse it away. Repeat shampoo/rinse until you feel clean. If it is cold outside, use a hair-drier, otherwise let it dry by itself.
- Algorithm:
  1. Wet hair
  2. Repeat until hair is clean
     - 2.1 rub in shampoo
     - 2.2 rinse shampoo away
  3. If cold outside
     - 3.1 use hair-drier
  4. Otherwise
     - 4.1 let the hair dry by itself

An algorithm is a precise description of a problem solution. It is often structured as a program $\Rightarrow$ easy to convert into a running program.

Algorithm: Prime Numbers

- Is N a prime number?
- Basic Idea: Check if N can be divided by any of the numbers in the interval 2 to $N-1$. If that is the case, N is **not** a prime number.
- Notice: N can be divided by $M \Rightarrow N \mod M = 0$
- Algorithm:

```
Test = 2
while N mod Test ≠ 0 do
    Test = Test + 1
end while
if N = Test then
    N is a prime number
else
    N is not a prime number
end if
```

Notice: The algorithm above can easily be implemented in Java (or any other language) if you know how to program.

Algorithms – Defintion

An **algorithm** is a step-by-step description of how to solve a problem.
In addition to being correct it should:

1. Give an unambiguous result
   $\Rightarrow$ only one result for each input
2. Be unambiguously presented
   $\Rightarrow$ a precise formulation that can't be misinterpreted
3. Terminate after a finite number of steps
   $\Rightarrow$ no infinite computations.

An infinite computation of π

$$\pi = 4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots)$$

- **Why Algorithms?**
  - Document problem solutions
  - Communicate problem solutions
  - Compare problem solutions (time complexity!)
  - Also, sketchy algorithms are a good preparation before

Problem: Sort a Deck

- Q: How do you sort a deck of cards?
- Algorithm:
  1. Sort in colours $\Rightarrow$ four piles
  2. Repeat for each pile (colour)
     - 2.1 sort by rank $2 < 3 < ... < K < A$
  3. Collect sorted piles to a deck of cards

Problem

- Not unambiguously presented
- How do you sort in colours?
- How do you sort by ranks?

Ask yourself: Does the algorithm describe an implementation?

However: This type of sketchy algorithm is often a good starting point. It can later be refined to a complete algorithm.

Algorithm Description Languages

- **Ordinary Text:**
  - Advantage: Everyone understands
  - Disadvantage: Often ambiguous ⇒ vague, not easy to implement
- **Programming Language**
  - Advantage: unambiguous, an implementation in itself
  - Disadvantage: requires knowledge about the language, contains non-relevant details. (For example, semi-colon, `System.out.println`, `Math.sin(x)`)
- **Pseudo Code**: A mix of ordinary text and programming language
  - Advantage:
    - Not coupled to a specific programming language
    - Easy to understand for a programmer
    - Often easy to implement
      ⇒ basically translate statement-by-statement to a given language

Pseudo Code - Two Examples

Problem: Can N be divided by 3?

Method 1: Structured ordinary text

1. Ask user for an integer. Denote this integer N
2. If N modulus 3 equals 0
   - 2.1 Inform user that N is dividable by 3
3. Otherwise
   - 3.1 Inform user that N is **not** dividable by 3

Method 2: Almost like a program

N : integer to be tested
if $N \mod 3 = 0$ **then**
    N is dividable by 3
else
    N is **not** dividable by 3
end if

Notice: There are no exact rules for pseudo code.

Pseudo Code

- A mix between programming language and ordinary text
- More precise than ordinary text,
  more general than programming language
- Programming language independent
- Often uses so-called **primitives** (`if`, `while`, `for`, `foreach`, `...`) and notations from mathematics and logic
- Must be able to handle the following constructs
  - Sequence of instructions
  - Selection (choice/alternative)
  - Iteration (repeat)
  - Simple arithmetics (plus, minus, ...)

  Theory: We don't need more constructs than these four
  $\Rightarrow$ the description is *Turing complete*
- Algorithms are often presented as procedures with parameters representing input data.

Problem Solving and Algorithms

Successful problem solving according to Polya[1]:

- Understand the problem.
- Develop a plan to solve the problem.
- Implement the plan.
- Reflect over the implementation.

Problem solving with programming:

- Understand the problem
- Create an algorithm
- Implement the algorithm
- Run the program and evaluate the result (Testing!)

[1] A Hungarian Mathematician

Stepwise Refinement (Divide and Conquer)

- Stepwise refinement: Larger problems are divided into smaller sub-problems which can, in turn, be divided to even smaller sub-problems.
- Gives structure to the problem solving.
- Gives smaller, easier to handle, sub-problems.
- Can give hints to where to divide a program, how to construct modules and so on.

Stepwise refinement and algorithm for Problem X

1. Subproblem 1
   1.1 Refinement of subproblem 1
   1.2 ...
2. Subproblem 2
   2.1 Refinement of subproblem 2
   2.2 ...

Ex: Stepwise Refinement

Problem: Do the dishes.

- A description of the problem on a high level:
  1. Rinse away the leftovers.
  2. Do the dishes.
  3. Clean up.
- Refinement of problem 2:
  - 2.1 Pour water.
  - 2.2 Add some detergent.
  - 2.3 Wash all the items.
- Refinement of problem 2.3:
  - 2.3.1 Repeat until no more items
    - 2.3.1.1 Take an item and put it in the water.
    - 2.3.1.2 Brush it.
    - 2.3.1.3 Rinse.
    - 2.3.1.4 Put it up to dry.

This method is a really good preparation before programming

Instance of a problem

- Most often we like to find a general solution to a problem. We say there is a *general* problem statement and a number of *instances* of the problem.
- Addition
  - Generally: What is the result of adding two integers?
  - Instance: What is 13+25?
- Sorting
  - Generally: Sort a list with names alphabetically
  - Instance: Sort [Lisa, Olle, Kalle, Anna] alphabetically.
- **My advice:**
  - Begin with a number of instances...
  - ...then attack the general problem

How do we recognize a good algorithm?

- We expect each algorithm to be *correct* ...
- ...but there might be more than one correct algorithm.
- Which one is the best?

Possible criteria:

- The algorithm is easy to understand and implement
  - simple
  - clearly written
  - well documented
  - ...
- The algorithm is *efficient*
  - uses resources efficiently, for example memory or network capacity
  - time efficient $\Rightarrow$ fast!

We will concentrate on time efficiency but that does NOT mean that the other criteria are not important.

Asymptotic Analysis

We would like to:

- Analyse algorithms without knowing on which computer they will execute
- Answer questions like "Which of these two algorithms are faster if the input size is big?".
- Answer questions like "How much will the computation time increase if the size of the input is multiplied by 2?"

We will achieve this by using *asymptotic analysis* and the big-oh notation $\Rightarrow$ a *Time Complexity* estimate.

Asymptotic Analysis $\Rightarrow$ Behaviour when input size is big.

Time Complexity (Introduction)

Time Complexity: An estimate of required computation time.

- Number of required computations often depend on input data
  - Find integer in a list ⇒ time depends on list size N
  - Check if N is a prime number ⇒ time depends on N size
  - Sort list ⇒ time depends on list size N
- We say that an algorithm have time complexity
  - $O(N)$ if computation time is proportional to N
  - $O(N^2)$ if computation time is proportional to N^2
  - $O(1)$ if computation time is constant
  - in general, $O(F(N))$ if computation time is proportional to $F(N)$
- $O(\dots)$ is pronounced *Big-Oh of* .... (Example Big-Oh of N-square.)
- or sometimes *Ordo of* ...
- **Basic assumption: Each simple computation takes time 1**
- Simple operations: `+`,`-`,`\`,`*`,%, assignment, ...

Time Complexity: Examples

- Print multiplication table for N $\Rightarrow O(N^2)$

```
public void printTable(int N) {   // O(N)
   for (int i=0;i<N;i++) {             // O(N)
      for (int j=0;j<N;j++)
         System.out.println(i*j);  // executed N*N times
   }
}                    //   ==> O(N^2)
```

- Search for X in array of size N $\Rightarrow O(N)$

```
public boolean search(int X, int[] arr) {
   for (int i=0;i<arr.length;i++) {        // O(N)
      if (arr[i] == X)   // executed N times // O(1)
         return true;
   }
   return false;
}     // ==> O(N)
```

Asymptotic Handling in Practise

Assume time $T(n) =$ in terms of input size n.

1. Constant factors do not matter.
2. In a sum, only the term that grows fastest is important.

- $T(n) = 3n \quad \Rightarrow O(n)$,
- $T(n) = 4n^4 - 45n^3 + 102n + 5 \quad \Rightarrow O(n^4)$,
- $T(n) = 16n - 3n \cdot \log_2(n) + 102 \quad \Rightarrow O(n \cdot \log_2(n))$,
- $T(n) = 9168n^{88} - 3n \cdot \log_2(n) + 5 \cdot 2^n \quad \Rightarrow O(2^n)$

- The $O(...)$ notation describes the behaviour when input size is big
- We are always interested in the *worst-case scenario*
  $\Rightarrow$ Not when we are finding an element at the first position in a list

Frequent Big-Oh Expressions

- $O(1)$ At most constant time, i.e. not dependent on the size of the input.
- $O(\log n)$ At most a constant times the logarithm of the input size.
- $O(n)$ At most proportional to n.
- $O(n \log n)$ At most a constant times n times the logarithm of n.
- $O(n^2)$ At most a constant times the square of n.
- $O(n^3)$ At most a constant times the cube of n.
- $O(2^n)$ At most exponential to n.

They are ordered from fastest ($O(1)$) to slowest ($O(2^n)$).

A 10 Minute Break

zzzzzzzzzzzzz ...

Sorting and Searching

- To *search* and *sort* are very common operations.
- **Searching:** Check if a given element N is in a list (or data collection).
- **Sorting:** Sort the elements in a list on a given property of the element.
  (alphabetically, by size, points on an exam and so on.)
- **This lecture**
  - Two simple sorting algorithms running in $O(N^2)$
  - One more advanced running in $O(N \log N)$
  - Two simple search algorithms
  - Reusable implementations

Selection Sort

- Basic idea: Assume a list with N elements
  - Find the smallest of the numbers and swap place with the first element
  - Find smallest of the $N-1$ last numbers and swap with second element
  - ...
  - Find the smallest of the $N-i+1$ last numbers and swap with the ith element
  - ...
  - Find the smallest of the two last numbers and swap places with the last but one element
- **Example:** Assume list `[ 6, 5, 9, -12, -2, -7 ]`

```
-12 is the smallest, swap with position 0 ==> [ -12, 5, 9, 6, -2, -
-7  is the smallest, swap with position 1 ==> [ -12, -7, 9, 6, -2,
-2  is the smallest, swap with position 2 ==> [ -12, -7, -2, 6, 9,
 5  is the smallest, swap with position 3 ==> [ -12, -7, -2, 5, 9,
 6  is the smallest, swap with position 4 ==> [ -12, -7, -2, 5, 6,
```

Algorithm for Selection sort

Algorithm: Let L be a list with index 1 to N

```
for each  i ∈ [1, N − 1] do
    p = i                          { i = position to be sorted}
    for each  j ∈ [i + 1, N] do
        if L[j] < L[p] then
            p = j                          {Smallest so far}
        end if
    end for
    temp = L[i]            {Swap smallest (p) and the first (i)}
    L[i] = L[p]
    L[p] = temp
end for
```

Time complexity: The inner loop is executed $N(N+1)/2$ times
$\Rightarrow O(N^2)$.
Better sorting algorithms can sort list in $O(N \cdot log(N))$
$\Rightarrow$ makes quite a difference for sizes $>$ 1 million

selectionSort (ArraySortSearch.java)

```java
public static int[] selectionSort(int[] arr) {
    int sz = arr.length;
    for (int i=0 ;i<sz-1; i++) {
        int update = i;              // position to update
        int min = update;            // initialize min position
        for (int j=update+1; j<sz; j++) { // remaining elements
            if (arr[j] < arr[min])
                min = j;                      // update min
        }

        /* Swap update and min */
        int tmp = arr[update];
        arr[update] = arr[min];
        arr[min] = tmp;
    }
    return arr;
}
```

Insertion Sort

- Basic assumption: Assume a list with N elements
  - Put the 2nd element in the right place among the two first
  - Put the 3rd element in the right place among the three first
  - Put the 4th element in the right place among the four first
  - ...
  - Put the Nth element in the right place among the N first
- **Basic idea:** Assume the first P elements are sorted ⇒ Move element P+1 to the left until you find a suitable place ⇒ repeated swaps
- **Example:** Assume the list `[ 6, -7, 9, -12, -2, 5 ]`

```
Put -7 among the 2 first ==>  [ -7, 6, 9, -12, -2, 5 ]
Put 9 among the 3 first ==>   [ -7, 6, 9, -12, -2, 5 ]
Put -12 among the 4 first ==> [ -12, -7, 6, 9, -2, 5 ]
Put -2 among the 5 first ==>  [ -12, -7, -2, 6, 9, 5 ]
Put 5 among the 6 first ==>   [ -12, -7, -2, 5, 6, 9 ]
```

Selection vs Insertion

- Selection Sort: Assume a list with N elements
  - Find the smallest number and swap with the first
  - Find the smallest among the $N - 1$ last numbers and swap with the second element
  - ...
  - Find the smallest among the two last numbers and swap with the last but one element
- Insertion Sort: Assume a list with N elements
  - Insert the 2nd element in the right place among the two first
  - Insert the 3rd element in the right place among the three first
  - ...
  - Insert the Nth element in the right place among the N first
- **Which one is the best?**
- Which is the fastest on an already sorted list?

Linear Search

- Basic Idea: Sequential search

```
/* Return true if element n in array.   */
public static boolean linearSearch(int[] arr, int n) {
    for (int i=0;i<arr.length;i++) {
        if (arr[i] == n)
            return true;
    }
    return false;
}
```

- We must check every element in the list
  $\Rightarrow O(N)$, where N is the list/array size.

Q: Do we have better algorithms?

A: No, not for an arbitrary list (in a single-core machine).

Binary Search

- Problem: Find n in list with N elements
- **Assumption: The list is sorted**
- Basic idea: Look at the middle element $m = arr[M]$
  - If $n = m$, return middle position M
  - If $n < m$, repeat search in [0,M-1]
  - If $n > m$, repeat search in [M+1,N]
- Each "search" halves the problem
  $\Rightarrow T(N) = T(N/2) + O(1)$
- n not in list $\Rightarrow$ empty list in next search

```
Find 8 i [1,3,5,7,8,9,10] ==> middle element is 7 ==>
Find 8 i [8,9,10] ==> middle element is 9 ==>
Find 8 i [8] ==> OK!
```

- Much faster than linear search
  $\Rightarrow$ Might be worth sorting the list if searched many times.

Binary Search

- Steps (time) required to search list of different sizes
  - Size: 1 $\Rightarrow$ Time $= 1$
  - Size: 2 $\Rightarrow$ Time $= 2$
  - Size: 4 $\Rightarrow$ Time $= 3$
  - Size: 8 $\Rightarrow$ Time $= 4$
  - Size: 16 $\Rightarrow$ Time $= 5$
  - Size: 32 $\Rightarrow$ Time $= 6$
  - ...
  - Size: 2^p $\Rightarrow$ Time $= p + 1$
- Thus, $N \propto 2^t$ (Size as a function of time)
- $\Rightarrow t \propto \log_2(N)$ (Time as a function of size)
- $\Rightarrow T(N) = O(\log_2(N))$

In general, an algorithm that halves the problem in a fix number of computations has time-complexity $O(\log_2(N))$

Recursive Binary Search

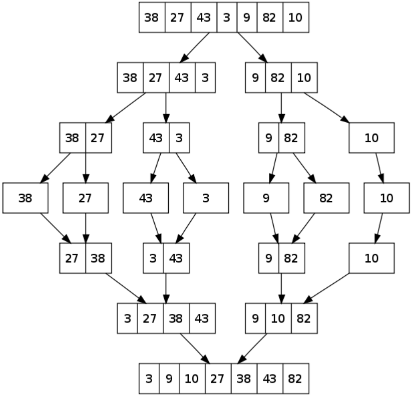
A recursive Java implementation (not dividing array into smaller arrays).

```
public static boolean recBinarySearch(int[] arr, int n) {
    int min = 0;
    int max = arr.length-1;
    return binSearch(arr, n, min, max);  // Start recursive search
}                                        // in interval [min,max]

private static boolean binSearch(int[] arr, int n, int min, int max) {
    if (max < min) return false;  // Not found!
    int mid = (min+max)/2;        // Mid position
    int mid_value = arr[mid];     // Mid element
    if (n == mid_value)  // Found!
        return true;
    else if (n < mid_value)
        return binSearch(arr,n,min,mid-1);  // Search left half
    else  // n > mid_value
        return binSearch(arr,n,mid+1,max);  // Search right half
}
```

Merge Sort

- Divide recursively the unsorted list into 2 sublists until each list contain 1 element
- Repeatedly Merge sublists to produce new sublists until there is only 1 sublist remaining
- Number of levels: $O(\log_2(N))$
- Work in each level: $O(N)$
- $\Rightarrow O(N \cdot \log_2(N))$



Should be compared to $O(N^2)$ for Selection Sort. Merge Sort is much faster for large lists. A recursive implementation is a bit tricky since we must work with indexes and avoid creating new lists for best performance.

Merge Sort (Outline)

Sketchy algorithm for recursive method `int[] mergeSort(int[] arr)`

1. If `arr` is of size 1 $\Rightarrow$ sorted $\Rightarrow$ return `arr`
2. Else, divide arr into two halves named `left` and `right`
3. Make calls `left = mergSort(left)` and `right = mergeSort(right)`
   $\Rightarrow$ the returned arrays `left` and `right` are now sorted
4. Merge `left` and `right` into one sorted array
   (Reuse `arr` for best performance!)
5. Return sorted array

Reusable Sorting

- We have solved the problem for integers ordered as lowest first
- How to do it for other types and/or other sorting criteria?
- Important: To sort Obj1 and Obj2 we must know when `Obj1 > Obj2`

- **Reusable Sorting using Comparable**
  1. The elements implement the Comparable Interface

```java
package java.lang;
public interface Comparable<T> {
    /* Returns a negative integer, zero, or a positive integer as
    * this object is less than, equal to, or greater than the
    * specified object.   */
    public int compareTo(T other);
}
```

  2. Implement: `public static void sort(Comparable[] arr)`
     $\Rightarrow$ Can be used on all arrays with `Comparable` elements.
  3. This should hold: `a.equals(b) ==> a.compareTo(b) = 0`

Name **implementing** Comparable

```
public class Name implements Comparable<Name> {
    private String first;
    private String last;

    // Skipping a few methods

    /* Implement interface Comparable */
    public int compareTo(Name other) {
            String other_first = other.getFirstName();
            String other_last = other.getLastName();

            if (other_last.equals(last))              // Using that strings
                return first.compareTo(other_first);   // are comparable
            else
                return last.compareTo(other_last);
    }
}
```

Selection Sort with `Comparable`

A reusable sorting algorithm

```
public static void sort(Comparable[] in) {  // Selection Sort
    int sz = in.length;
    for (int i=0;i<sz-1;i++) {
        int update = i;            // position to update
        int min = update;          // initialize min
        for (int j=update+1;j<sz;j++) { // remaining elements
            if (in[j].compareTo(in[min]) < 0)
                min = j;
        }
        /* Swap update and min */
        Comparable tmp = in[update];
        in[update] = in[min];
        in[min] = tmp;
    }
}
```

Notice

- This method can sort any array containing elements impementing Comparable

Sort/Search in Practice

- The Java class library has very good support for sorting and searching.
- String, Integer, Double, ... implements the Comparable interface
- The helper class `java.util.Arrays` has methods for arrays.
- The helper class `java.util.Collections` has methods for lists.
- You will find:
  - Method: `binarySearch()` – search in a sorted list/array
  - Method: `sort()` – sorts a list/array
- Both assume that the elements are implementing the `Comparable` interface.
- They can also take a Comparator lambda expression (see Lecture 3)
- Method `sort()` uses a variant of the *Merge sort* algorithm.
- Searching in an unsorted list is done using the method `contains()`, searching in an unsorted array is always done manually (approx. four lines of code).