Algorithms

1 What is an algorithm?

Definition
An algorithm is a finite set of precise instructions for performing a computation, or, more generally, for solving a problem

2 Vocabulary

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Input values, usually from a specified set</td>
</tr>
<tr>
<td>Output</td>
<td>Solution(s) to the problem at hand</td>
</tr>
<tr>
<td>Correctness</td>
<td>An algorithm should produce the correct output values for all possible input values</td>
</tr>
<tr>
<td>Generality</td>
<td>An algorithm should be applicable for all problems of the desired form, not just for a particular set of input values</td>
</tr>
<tr>
<td>Finiteness</td>
<td>An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input</td>
</tr>
<tr>
<td>Time complexity</td>
<td>Number of operations used by the algorithm when the input has a particular size</td>
</tr>
<tr>
<td>Pseudocode</td>
<td>An informal language that helps programmers develop algorithms</td>
</tr>
</tbody>
</table>
3 Algorithm 1: the sieve of Eratosthenes

3.1 Finding all prime numbers up to a number $n$

The sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to any given limit.

It does so by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2. Once all the multiples of each discovered prime have been marked as composites, the remaining unmarked numbers are primes.

To find all the prime numbers less than or equal to a given integer $n$ by Eratosthenes’ method:

1) Create a list of consecutive integers from 2 through $n$: (2, 3, 4, ..., $n$).

2) Initialize an integer $p$ to 2, the smallest prime number.

3) Enumerate the multiples of $p$ and mark them in the list (these will be $2p$, $3p$, $4p$, ...). Note that $p$ itself should not be marked.

4) Find the smallest number in the list greater than $p$ that is not marked. If there was no such number, stop. Otherwise, let $p$ now equal this new number (which is the next prime), and repeat from step 3.

When the algorithm terminates, the numbers remaining not marked in the list are all the primes below $n$. Some notes / refinement:

1) Some numbers may be marked more than once (for example, 15 will be marked both for 3 and 5).

2) In step 4, the algorithm is allowed to terminate when $p$ is greater that $\sqrt{n}$. We will show this later.

Example: find all prime numbers below 100

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
</tr>
<tr>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
</tr>
<tr>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
</tr>
<tr>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
</tr>
<tr>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
</tr>
<tr>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
</tr>
</tbody>
</table>

The prime numbers up to 100 are therefore: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, and 97.
3.2 Algorithm

The sieve of Eratosthenes can be implemented on a computer using the following pseudocode:

**Algorithm 1** Sieve of Eratosthenes

**Input:**
an integer N

**Initialize:** A is an array of Boolean values all set to true

for i = 2, 3, 4, . . . , not exceeding $\sqrt{n}$ do
  for j = 1, 2, 3, . . . with i × j not exceeding n do
    A[i × j] := false
  end for
end for

**Output:** all i such that A[i] is true.

Is this algorithm valid? We need to justify that (i), A[n] is true if and only if n is prime, and (ii), we only need to check for multiples of primes that are smaller than $\sqrt{n}$.

a) **We only need to check for multiple of prime numbers that are smaller than $\sqrt{n}$.**

This is a consequence of the following result:

**Property 1:** Let a and b be two integers. If $n = ab$, then $a \leq \sqrt{n}$ or $b \leq \sqrt{n}$

**Proof.** Let p be the proposition “$n = ab$”, and let q be the proposition $a \leq \sqrt{n}$ or $b \leq \sqrt{n}$. We use a proof by contradiction, namely we assume that p is true AND q is false.

Since q is false, we know that

\[ a > \sqrt{n} \quad (1) \]
\[ b > \sqrt{n}. \quad (2) \]

As $\sqrt{n}$ is positive, we have that both $a > 0$ and $b > 0$. We multiply (1) with b and (2) with $\sqrt{n}$:

\[ ab > b\sqrt{n} \quad (3) \]
\[ b\sqrt{n} > n. \quad (4) \]

By transitivity, we get that $ab > \sqrt{n}$, but we also have $ab = n$ as p is true. We have reached a contradiction. Therefore the original implication is true.

This property can be interpreted as follows: a composite number n will be the product of two integers a and b. We are then guaranteed that one of these two numbers is smaller than $\sqrt{n}$. 


b) \( A[n] = 1 \) if and only if \( n \) is prime.

**Proof.** We need to prove a biconditional between two propositions:
- \( P(n) : A[n] \) is True
- \( Q(n) : n \) is prime

We prove the biconditional by proving separately \( P(n) \to Q(n) \) and \( Q(n) \to P(n) \)

a) To show \( Q(n) \to P(n) \) we use an indirect proof. Let us assume \( P(n) \) is false, namely that \( A[n] \) is false. However, the algorithm only sets \( A[n] = \text{false} \) when \( n \) is of the form \( i \times j \), where \( i \) is a prime number. Therefore \( n \) is composite, and \( Q(n) \) is false. This validates that \( Q(n) \to P(n) \).

b) To show \( P(n) \to Q(n) \) we use an indirect proof. Let us assume that \( Q(n) \) is false, namely that \( n \) is composite. According to the fundamental theorem of arithmetics, \( n \) can be written as \( n = q_1q_2\ldots q_k \), where \( q_i \) are prime numbers. Let \( q_1 \) be the smallest of those numbers. We define \( a = q_1 \) and \( b = q_2\ldots a_k \). Then \( n = ab \), i.e. \( n \) is a multiple of \( a \). Based on property 1, \( a \leq \sqrt{n} \). Since \( a \) is prime, based on a), \( A[a] \) is true. All multiples of \( a \) are then set with their labels zero, and therefore \( A[n] \) is false, i.e. \( P(n) \) is false. This validates that \( P(n) \to Q(n) \).

## 4 Searching algorithms

**Definition:** Searching algorithms solve the problem of locating an element in a list.

This problem occurs in many contexts (think for example searching a name in a phone book...).

The simplest way to solve this problem is to test each element of the list: **linear search.**
4.1 Linear search

Algorithm 2 Linear search

Input:
N: the number of elements in the list (integer)
a_1, a_2, \ldots, a_N: the elements of the list
x: the element to be located

Initialize: Index ← 0
for (k = 1, k ≤ N, Step = 1) do
    if (a_k == x) then
        Index ← k
        return
    end if
end for

Output: Index of x in the list.

What is the complexity of this algorithm. It depends!

- If x = a_1, we only need one comparison
- If a = a_N, we need N comparisons

So very different behaviors! We define two different types of complexity:

| Worst case complexity: Largest number of operations needed to solve the given problem of a specified size |
| Average case complexity: Average number of operations needed to solve the given problem of a specified size |

**Worst case complexity for linear search:** The worst case occurs when x is not in the list in which case the loop is fully executed. As there are 2 comparisons per step (one that compares x to a_k, and one that compares k to N), the total number of comparisons in the worst case is 2N.

**Average case complexity for linear search:** if
- x = a_1: the number of comparisons is 2
- x = a_2: the number of comparisons is 4
  
  ... 

- x = a_k: the number of comparisons is 2k
- x = a_N or x not in the list: the number of comparisons is 2N
The average number of comparisons is therefore:

\[
S = \frac{2 + \ldots + 2N}{N} = 2 \frac{1 + \ldots N}{N} = \frac{2 N(N + 1)}{2} = N + 1
\]

The average number of comparisons is therefore \(N + 1\).

Linear search, however, is not optimal, especially in the case that the list to be searched is ordered.

### 4.2 Binary search

To explain binary search, let us consider an example. Let the list of elements to be searched be:

\[
S = \{1, 3, 7, 11, 17, 23, 32, 48, 49, 51, 53, 60\}
\]

and let \(x = 53\) be the element to be searched. If we use a linear search to do this search, the element will be found after 11 trials, and therefore the linear search requires 22 comparisons.

The binary search proceeds by dichotomy: we divide the set into two groups of approximately equal length. Since \(|S| = 12\), we pick the 6-th element as pivot, namely 23:

\[
S_1 = \{1, 3, 7, 11, 17, 23, 32, 48, 49, 51, 53, 60\}
\]

We compare 53 with this pivot. As 53 is bigger than 23, we only need to look at the right side of 23. There are six elements there, \(\{32, 48, 49, 51, 53, 60\}\) and we pick a new pivot as the third of this element, 49:

\[
S_2 = \{32, 48, 49, 51, 53, 60\}
\]

We compare 53 with this pivot. As 53 is bigger than 49, we only need to look at the right side of 51. There are three elements there, \(\{51, 53, 60\}\) and we pick a new pivot as the third of this element, 53:

\[
S_2 = \{51, 53, 60\}
\]

We compare 53 with this pivot: we are done! We only needed three comparisons, plus the three operations needed to generate the pivots, which is much faster than the linear search.
Algorithm 3 Binary search

Input:
N: the number of elements in the list (integer)
a_1, a_2, \ldots, a_N: the elements of the list
x: the element to be located

Initialize:
Index ← 0
Left ← 1
Right ← N

while (Left ≤ Right) do
    Center ← floor \left(\frac{Left + Right}{2}\right)
    if (a_{Center} == x) then
        Index ← Center
        return
    else if (a_{Center} < x) then
        Right ← Center − 1
    else
        Left ← Center + 1
    end if
end while

Output: Index of x in the list.

What is the worst case complexity of the binary search?

Each step of the binary search requires 2 comparisons, plus 1 operation (taking the floor). For simplicity, we assume that the total cost of one step is 3. How many steps do we need?

Let us assume that \( N = 2^k \). Note that otherwise we can find \( l \) such that \( 2^l \leq N < 2^{l+1} \) and we set \( k = l \). Then \( k = \log_2(N) \). At the first step of the binary search, \( x \) is compared with the middle of the list of size \( N \). Either \( x \) is found, or at the next step, it is searched into a list with size \( 2^{k-1} \). The procedure is then repeated until the list is of size 1:

\[
2^k \rightarrow 2^{k-1} \rightarrow 2^{k-2} \ldots 2^0
\]

Hence, after at most \( k \) steps, we know if \( x \) is not in the list, or if we have located it. Therefore, in the worst case scenario, we need \( 3 \log_2(N) \) operations.

Example: To find a name in a phone book containing 250 million entries may take 250 million steps using linear search, but only at most 28 steps using binary search.