COMP26121 - Algorithms and Imperative Programming

November 26, 2017

1 Algorithm Analysis

Scalability: the ability of a system to gracefully accommodate growing sizes of input/workloads.
Algorithm: a step-by-step mechanical procedure for performing some task in a finite amount of work.
Data Structure: is a systematic way of organising and accessing data.
We analyse algorithms using their **running time**, which is usually dependent on the size of the **input**, the hardware environment and the software environment.
Pseudo-code: mixture of natural language and high-level programming constructs that describe the main ideas behind a generic implementation of a data structure or algorithm. It includes:

- Expressions.
- Method declarations.
- Decision structures: if-then-else.
- While and For Loops.
- Array indexing.
- Method calls.
- Method returns.

Primitive operation: low-level instruction with an execution time that depends on the hardware and software environment but is nevertheless constant.

We just **count** the number of operations.

- Assigning value to a variable.
- Calling a method.
- Performing an arithmetic operation.
- Comparing two numbers.
- Indexing into an array.
- Following an object reference.
- Returning from a method.

Random Access Machine: CPU connected to memory cells, an arbitrary memory cell can be accessed with a single primitive operation.
An **average case analysis** is challenging since it requires most of the times heavy maths and probability theory. We then use a **worst-case analysis** which is easier since we only have to identify the worst-case input. The best and worst cases are two extreme ones. The average case usually coincides with the worst case, but there are times it is very different from both of the extreme ones (see Quick-Sort).
Recursion: a procedure P is allowed to make calls to itself as a subroutine, provided these calls to P are for solving sub-problems of smaller size. Recursion needs a **base case** which can be solved without using recursion. To analyse its running time we use a **recurrence equation**.

Big-Oh notation: Let $f(n)$ and $g(n)$ be functions mapping non-negative integers to real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $C > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq Cg(n)$ for every integer $n \geq n_0$. Thus, $O(f)$ denotes a **set of functions**. Any polynomial $a_k n^k + a_{k-1} n^{k-1} + ... + a_0$ will always be $O(n^k)$.

When comparing growth-rates functions, small values and linear factors are ignored, as we are interested in how functions behave in the long run.

Big-Omega notation: Let $f(n)$ and $g(n)$ be functions mapping non-negative integers to real numbers. We say that $f(n)$ is $\Omega(g(n))$ if $g(n)$ is $O(f(n))$; that is, there is a real constant $C > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq Cg(n)$ for $n \geq n_0$.

Describes *greater than or equal* behaviours. $f \in \Omega(g)$ if and only if $g \in O(f)$.

Big-Theta notation: Likewise, $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$; that is, there is a real constant $C' > 0$ and $C'' > 0$, and an integer constant $n_0 \geq 1$ such that $C'g(n) \leq f(n) \leq C''g(n)$, for $n \geq n_0$.

Describes *asymptotically equals* behaviours. $\Theta(f) = O(f) \cap \Omega(f)$.

The division between polynomial-time and exponential-time algorithms is considered a robust measure of tractability.

Little-Oh notation: *strictly less than* asymptotically.

Little-Omega notation: *strictly greater than* asymptotically.

The difference between *Big-Oh* and *Little-Oh* is that $f(n)$ is $O(g(n))$ if **there exist** constant $C > 0$ and $n_0 \geq 1$ such that $f(n) \leq Cg(n)$, for $n \geq n_0$; whereas $f(n)$ is $o(g(n))$ if **for-all** constants $C > 0$ there is a constant n_0 such that $f(n)$ becomes insignificant compared to $g(n)$ as n grows towards infinity.

1.1 Mathematical Review

Summation: $\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + ... + f(b)$.

$\sum_{i=0}^{n} a^i = 1 + a + a^2 + ... + a^n = \frac{1-a^{n-1}}{1-a}$ with $n \geq 0$ and $a > 0$.

$\sum_{i=1}^{n} i = 1 + 2 + ... + (n-1) + n = \frac{n(n+1)}{2}$ with $n \geq 1$.

Logarithm: $\log_b a = c$ if $a^c = b$.

Floor: $\lfloor x \rfloor$ = the largest integer less than or equal x.

Ceiling: $\lceil x \rceil$ = the smallest integer greater than or equal x.

Induction: showing that for any $n \geq 1$ there is a finite sequence of implications that starts with something known to be true and ultimately leads to showing that that is true.

Loop invariant: a predicate P which is true at the start and at the end of each execution of the body of a while-loop.

Probability space: is a sample space S together with a probability function Pr that maps subsets of S to real numbers in the interval $[0, 1]$. Each subset A of S is an *event*, and Pr is assumed to possess the following properties with respect to events defined from S:

1. $Pr(\emptyset) = 0$.
2. $Pr(S) = 1$.
3. $0 \leq Pr(A) \leq 1$ for any $A \subseteq S$.
4. If $A, B \subseteq S$ and $A \cap B$, $Pr(A \cup B) = Pr(A) + Pr(B)$.

Independence: $Pr(A \cap B) = Pr(A)Pr(B)$.

Mutual Independence: $Pr(A_{i1} \cap A_{i2} \cap ... \cap A_{ik}) = Pr(A_{i1})Pr(A_{i2})...Pr(A_{ik})$ for any subset $A_{i1}, A_{i2}, ..., A_{ik}$ of the collection of events $A_1, A_2, ..., A_n$.

Conditional Probability: $Pr(A|B) = \frac{Pr(A \cap B)}{Pr(B)}$ assuming $Pr(B) > 0$.

1.2 Amortised Analysis

Amortisation: gives us an *average-case analysis* without using any probability.

Amortised running-time: most case running time of the series of operations divided by the number of operations.

Accounting method: an amortised scheme of credits and debits for keeping track of the running time of the different operations in the series. An operation is changed a number of cyber-dollars equal to the number of primitive operations performed.

Potential function: we associate with out structure a value, Φ, which represents the current energy state of our system. Each operation that we perform will contribute some additional

amount (**amortised time)** to Φ but also extracts value from Φ in proportion to the amount of time spent.
Extendable array strategy: after we perform array replacement, our new array allows us to add n new elements to the table before the array must be replaced again. Efficient even if at first it does not seem.

2 Binary Search Trees

Algorithm 1 `BinarySearch`$(A, k, low, high)$

Input: An ordered array A, storing n items, whose keys are accessed with method $key(i)$ and whose elements are accessed with method $elem(i)$; a search key k and integers `low` and `high`.
Output: An element of A with key k and index between `low` and `high`, if such an element exists, and otherwise the special element `null`.

```
if low > high then
    return null
else
    mid ← ⌊(low+high)/2⌋
    if k = key(mid) then
        return elem(mid)
    else if k < key(mid) then
        return BinarySearch(A, k, low, mid − 1)
    else
        return BinarySearch(A, k, mid + 1, high)
```

The running time is proportional to the number of recursive calls performed. The number of remaining candidates is reduced by at least one half with each recursive call.
In the worst case, the recursive calls stop when there are no more candidate items. Hence, the maximum number of recursive calls performed is the smallest integer such that $\frac{n}{2^m} < 1$. Since $m = \lfloor \log n \rfloor + 1$, this implies that `BinarySearch`$(A, k, low, high)$ runs in $O(\log n)$ time.
The **space requirement** is optimal, $\Theta(n)$ since we have to store n objects.
Binary Search Tree: binary tree in which each internal node v stores an element e such that the elements stored in the left subtree of v are less than or equal to e, and the elements stored in the right subtree of v are greater than or equal e. External nodes store no elements, they could be `null`.
An **in-order traversal** visits the elements in the tree in non decreasing order. The searching is done comparing the element we are looking for with the element at a node.

Algorithm 2 `TreeSearch`(k, v)

Input: A search key k, and a node v of a binary search tree T.
Output: A node w of the subtree $T(v)$ of T rooted at v, such that either w is an internal node storing key k or w is the external node where an item with key k would belong if it existed.

```
if v is an external node then
    return v
if k = key(v) then
    return v
else if k < key(v) then
    return TreeSearch(k, T.leftChild(v)
else
    return TreeSearch(k, T.rightChild(v)
```

H executes a constant number of primitive operations for each node it traverses in the tree. The number of nodes is bounded by $height + 1$ hence the algorithm runs in $O(height)$. The height can be as big as n but it usually is smaller; the best height we can achieve is $\lceil \log(n + 1) \rceil$, the algorithm runs in $O(\log n)$.

3 Merge-Sort and Quick-Sort

Inverted file: allows a search engine to quickly return a list of the documents that contain a given keyword.
Divide and Conquer

1. **Divide:** divide the input data into two or more subsets.
2. **Recur:** recursively solve the subproblem.
3. **Conquer:** take solutions of subproblem and merge them together.

3.1 Merge-Sort

Algorithm 3 `merge`(S_1, S_2, S)

Input: Two arrays, S_1 and S_2, of size n_1 and n_2, sorted in non-decreasing order, and an empty array, S, of size at least $n_1 + n_2$.
Output: S, containing the elements from S_1 and S_2 in sorted order.

```
i ← 1
j ← 1
while i ≤ n and j ≤ n do
    if S1[i] ≤ S2[j] then
        S[i + j − 1] ← S1[i]
        i ← i + 1
    else
        S[i + j − 1] ← S2[j]
        j ← j + 1
while i ≤ n do
    S[i + j − 1] ← S1[i]
    i ← i + 1
while j ≤ n do
    S[i + j − 1] ← S2[j]
    j ← j + 1
```

`merge` has three while loops, inside each loop $O(1)$ time is taken. At each execution one element is not considered anymore. Hence, running time is $O(n_1 + n_2)$. The time spent at a node is $O(\frac{n}{2^i})$ and since the tree has exactly 2^i nodes the time spent at depth i is $O(2^i \frac{n}{2^i}) = O(n)$. Since the time spent at each node of the $\log(n+1)$ levels is $O(n)$ merge-sort runs in $O(n \log n)$ time when n is a power of 2.
When n is not a power of 2 we have this equation:

$$t(n) = \begin{cases} b & \text{if } n = 1 \text{ or } n = 0, \\ t(\lceil \frac{n}{2} \rceil) + t(\lfloor \frac{n}{2} \rfloor) + cn & \text{otherwise.} \end{cases}$$

A *closed form* characterisation (where $t(n)$ does not appear on both side of the equation) is:

$$\begin{aligned} t(n) &= 2^{\log n} t\Big(\frac{n}{2} \log n\Big) + (\log n)cn \\ &= nt(1) + cn \log n \\ &= nb + cn \log n \end{aligned}$$

Hence the running time is $O(n \log n)$.

3.2 Quick-sort

Quick-Sort:

1. **Divide:** select element x (pivot). Split the elements in:
   - L: elements in S less than x.
   - E: elements in S equal to x.
   - G: elements in S greather than x.
2. **Recur:** sort sequences L and G.
3. **Conquer:** put the elements in S in order by first inserting the elements of L, then those of E, then those of G.

The height of the tree can be $n-1$ in the worst case (when the list is already sorted). In this case we have to sum up the time spent at each node, which his input size is $n-1$ (i is the level in the recursion).
When the height is $n-1$ then the running time is $O(\sum_{i=0}^{n-1} n-i)$ which is $O(\sum_{i=1}^{n} i) = O(n^2)$.
When the splitting of S makes L and G of roughly the same size we have that $s_i = n-(1+2+2^2+...+2^{i-1}) = n-(2^i-1)$ which gives a running time of $O(n \log n)$ time.
Randomised Quick-Sort: if we *select the pivot randomly* we have a better chance of getting a $O(n \log n)$ running time because the expected number of innovation we must make until this occurs is $2 \log_{\frac{3}{4}} n$ and since we spend $O(n)$ at each level of the tree the total running time is $O(n \log n)$.
We can modify quick-sort in order to turn it in a **in-place** sorting algorithm: one which does not require additional memory for sorting the objects.

Algorithm 4 `inPlacePartition`(S, a, b)

Input: An array S of distinct elements; integers a and b such that $a \leq b$.
Output: An integer l such that the subarray $S[a \dots b]$ is partitioned into $S[a \dots l-1]$ and $S[l \dots b]$ so that every element in $S[a \dots l-1]$ is less than each element in $S[l \dots b]$.

```
Let r be a random integer in [a, b]
Swap S[r] and S[b]
p ← S[b]                                              // the pivot
l ← a                                                 // l will scan rightward
r ← b − 1                                             // r will scan leftward
while l ≤ r do                                        // find an element larger than the pivot
    while l ≤ r and S[l] ≤ p do
        l ← l + 1
    while r ≥ l and S[r] ≥ p do                       // find an element smaller than the pivot
        r ← r − 1
    if l < r then
        Swap S[l] and S[r]
Swap S[l] and S[b]                                    // put the pivot into its final place
return l
```

Algorithm 5 `inPlaceQuickSort`(S, a, b)

Input: An array S of distinct elements; integers a and b.
Output: The subarray $S[a \dots b]$ arranged in non decreasing order.

```
if a ≥ b then
    return                                            // subrange with 0 or 1 elements
l ← inPlacePartition(S, a, b)
inPlaceQuickSort(S, a, l − 1)
inPlaceQuickSort(S, l + 1, b)
```

We are using a small constant amount of space to store a, b, l, r. We also occupy extra space due to the recursive calls, this can become as deep as $\Theta(n)$.

To fix this we change the algorithm and we make it **tail recursive**. In this way the depth of the recursive stack is never more than $O(\log n)$.

Algorithm 6 `CorrectInPlaceQuickSort`(S, a, b)

Input: An array S of distinct elements; integers a and b.
Output: The subarray $S[a \dots b]$ arranged in non decreasing order.

```
while a < b do
    l ← inPlacePartition(S, a, b)
    if l − a < b − l then                          // first subarray is smaller
        CorrectInPlaceQuickSort(S, a, l − 1)
        a ← l + 1
    else
        CorrectInPlaceQuickSort(S, l + 1), b
        b ← l − 1
```

The running time of any comparison-based algorithm for sorting an n-element sequence is $\Omega(n \log n)$ in the worst case. This is true because the sorting can be described using a decision tree in which the path from the root to each external node represents a permutation of the array. The number of permutation is hence $n!$. Thus, the height is $\log(n!)$ and $\log(n!) \geq \log(\frac{n}{2})^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2}$ which is $O(n \log n)$.

4 String Algorithms

Substring: given a m-character string P, is a string of the form $P[i]P[i+1]P[i+2]...P[j]$ for some $0 \leq i \leq j \leq m-1$. If $i > j$ then $P[i...j]$ is the `null` string, which has length 0. A *proper substring* is one where either $i > 0$ or $j < m-1$.
Prefix: any substring of the form $P[0...i]$ for $0 \leq i \leq m-1$.
Suffix: any substring of the form $P[i...m-1]$ for $0 \leq i \leq m-1$.

The running time is $O(\log n)$, where n is the number of elements in the array.

Algorithm 7 `DictLookup`(t, D)

Input: A word t, a dictionary D of n words.
Output: A string which confirm that the word has been found.

```
beginp ← 0, ptr ← n/2, endp ← n
while beginp < endp do
    val ← strncmp(t, D[ptr])
    if val = 0 then
        print "Word found", exit
    if val > 0 then
        beginp ← ptr + 1
    else
        enp ← ptr
    ptr ← (beginp + endp)/2
```

For an input of n words and a dictionary of size m, it will run in worst case $O(n \log m)$ time.

Algorithm 8 `SpellChecker`(T, D)

Input: A text file T of n words, a dictionary D of m words.
Output: A string which ask whether a word has been misspelled.

```
Read the text file and put words into an array, word[]
for i ← 1 to n do
    found? ← DictLookup(word[i], D)
    if ∄found? then
        print word[i] "misspelled?"
```

Pattern matching: we want to find whether a patter string P is a substring of a text string T. That is $P = T[i \dots i+m-1]$.

A **brute force** approach pattern is a technique for algorithm design when we have something we wish to search for or when we wish to optimise some function and we can afford to spend a considerable amount of time optimising it.

Algorithm 9 `BruteForceMatch`(T, P)

Input: Strings T(text) with n characters and P(pattern) with m characters.
Output: Starting index of the first substring of T matching P, or -1.

```
for i ← 0 to n − m do                                  // for each candidate index in T
    j ← 0
    while j < m and T[i + j] = P[j] do
        j ← j + 1
    if j = m then
        return i
return −1
```

This algorithm simply tests *all the possible placements* of P relative to T. The running time of this method is $O(nm)$, which turns into $O(n^2)$ when $m = \frac{n}{2}$.
Lexicon Matching: given a set $L = \{P_1, P_2, \ldots, P_l\}$, called lexicon, of l different pattern strings and a text string T we want to find all places where a pattern P_i is a substring of T.
Uniform Hash $h(x)$: for any pattern P_k in L, the number of other patterns P_j with $j \neq k$, such that $h(P_i) = h(P_j)$, is $O(1)$.

Algorithm 10 `HashMatch`(L, T)

Input: A set, $L = \{P_i, \ldots, P_l\}$, of l pattern strings, each of length m and a text string T of length m.
Output: Each pair, (i, k), such that a pattern, P_k, in L appears as a substring of T starting at index i.

```
Let H be an initially empty set of key-value pairs (with hash-value keys)
Let A be an initially empty list of integer pairs (for found matches)
for k ← 1 to 1 do
    Add the key-value pair, (h(P_k), k) to H
for i ← 0 to n − m do
    f ← h(T[i ... i + m − 1])
    for all key-value pair, (f, k), with key f in H do  // check P_k against T[i ... i + m − 1
        j ← 0
        while j < m and T[i + j] = P_k[j] do
            j ← j + 1
        if j = m then
            Add (i, k) to A                             // a match at index i for P_k
return A
```

Assume computing $h(x)$ takes $O(m)$ for any string x of length m. Computing the hash-value keys and inserting them in a hash table takes $O(lm)$, or $O(lm \log l)$ if we use a tree. Computing the hash value of the substring in T requires $O(nm)$ time. The while loops require $O(lnm)$ time. Hence the worst-case (when all the hash values of the patterns are identical) is $O(lnm + l \log l)$. If we assume that the hash function is uniform the algorithm runs in $O(lm + nm)$ time (if H is an hash table).
This is known as the **Rabin-Karp** algorithm, and here we present another version. Instead of the hashing function here we use modular arithmetic in order to find an occurrence of P in T. We think of the elements of the alphabet Σ as digits in a base-b numeral, where $b = |\Sigma|$. We then encode P as the number $P[0] \cdot b^{m-1} + \cdots + O[m-1] \cdot b^0$ and $T[i, \ldots, i+m-1]$ as $T[i] \cdot b^{m-1} + \cdots + T[i+m-1] \cdot b^0$. Since these numbers are quite big we use some prime constant q to turn everything in $\mod q$. This holds because if $T[i, \ldots, i+m-1] \neq P \mod q$ we know that we do not have a match at shift i. On the other hand, if $T[i, \ldots, i+m-1] = P \mod q$ we then simply check explicitly that $T[i, \ldots, i+m-1] = P$.

Algorithm 11 `RabinKarp`(T, P, q, b)

Input: A text string T, a pattern P of length m, a modulus prime p and the size of the alphabet b.
Output: The indexes of the occurrences of P in T, if any.

```
I ← ∅
m ← |P|
t ← T[0] · b^{m−1} + ··· + T[m − 1] · b^0  mod q
p ← P[0] · b^{m−1} + ··· + P[m − 1] · b^0  mod q
i → 0
while i ≤ |T| − m do
    if p = t then
        j ← 0
        while P[j] = T[i + j] and j < |P| do
            j ← j + 1
        if j = |P| then
            I ← I ∪ {i}
    t ← (t − T[i] · b^{m−1}) · b + T[i + m]  mod q        // This computes T[i + 1, ..., i + m]
    i ← i + 1
return I
```

5 Cryptography

Goals:

- **Data Integrity:** information should not be altered without detection.
- **Authentication:** individuals and organisations that are accessing or communicating sensitive information must be correctly identified.
- **Authorisation:** agents that are performing computations involving sensitive information must be authorised to perform them.
- **Non-repudiation:** in transactions that imply a contract, the parties that have agreed to that contract must not have the ability of backing out of their obligations without being detected.
- **Confidentiality:** sensitive information should be kept secret from individuals who are authorised to see that information.

5.1 Euclid's GCD Algorithm

We know that $a = qb + r$ and $r = qb - a$, hence $\gcd(a, b) = \gcd(b, r)$.

Algorithm 12 `EuclidGCD`(a, b)

Input: Non-negative integers a, b (Assume $0 \neq a \geq b$).
Output: $\gcd(a, b)$

```
if b = 0 then
    return a
else
    r ← a  mod b
    return EuclidGCD(b, r)
```

The number of operation is proportional to the number of sensitive calls. We know that $a_{i+2} < \frac{1}{2}a_i$, hence the size of the first argument decreases by half with every iteration (recursive call). Hence the running time of `EuclidGCD` is $O(\log \max(a, b))$. If we let n denote the input size in bits, we characterise the performance of the algorithm as using $O(n)$ operations.

`EuclidGCD` is **linear in the size of the input**.

In order to compute $a^p \mod n$ we can follow a naive approach which is $O(n^2)$ in the size of the input.

5.2 Modular Arithmetics

When we perform aithmetic $\mod k$ we can stay within the numbers $0, \ldots, k-1$.
If our modulus is a prime number p then there is a *unique* number b such that $a \cdot b = b \cdot a = 1 \mod p$, this number b is called the **inverse** of $a \mod p$.

For example, $3 \cdot 5 = 5 \cdot 3 = 1 \mod 7$.

Algorithm 13 `FastExponentiation`(a, p, n)

Input: Integers a, p, n.
Output: $r = a^p \mod n$

```
s ← 1
while p > 0 do
    if e is odd then
        s ← s · a mod k
    a ← a^2 mod k
    b ← ⌊e/2⌋
return s
```

For $1 \leq a < p$, the smallest k such that $a^k = 1 \mod p$ divides $p-1$, and this is always true for p prime. Hence, $a^(p-1) \mod p$ is always 1, as stated in Fermat's Little Theorem.
The size of a positive integer n is $\lfloor \log_1 0n \rfloor + 1$.
The size of the operand of each multiplication and modulo operation is never more then $2\lceil \log_2 n \rceil$ bits.
The number of recursive calls and arithmetic operations is $O(\log p)$; since p is represented in binary using $O(\log p)$ bits the number of arithmetic operation is **linear** in the size of the exponent p in bits.
There is always some number a such that the various powers g^i cover the whole of $\{1, 2, \ldots, p-1\}$. In this case g is a **primitive root** $\mod p$.
Modular exponentiation is an example of a **one-way function** - easy to compute, hard to invert.
The existence of one-way functions is equivalent to the conjecture $P \neq NP$.

5.3 ElGamal Cryptosystems

Discrete logarithm: integer x such that $b^x = y \mod n$. This is really hard to compute.
Encryption phase: the **cypher-text** C is the pair (a, b)

$$a \leftarrow g^k \mod p$$

$$b \leftarrow My^k \mod p$$

where k is a random integer, M is the message (consider it as an integer), x is the **private key** and (p, g, y) is the **public key** where $y = g^x \mod p$.
Decryption phase: $M \leftarrow b(a^x)^{-1} \mod p$ where $(a^x)^{-1}$ is the inverse of a^x in Z_p.

$$\begin{aligned} b(a^x)^{-1} \mod p &= My^k(a^x)^{-1} \mod p \\ &= Mg^{xk}(g^{xk})^{-1} \mod p \\ &= M \end{aligned}$$

Both encryption and decryption in ElGamal systems take $O(\log n)$ operations where n is the modulus.

6 Mock Exam

6.1 Sorting

- *Question a goes here*

Algorithm 14 `merge-sort`(L, a, b) - Pseudocode

Input: A list of integers L, an initial index a and a final index b (Assume L indexes 0 to n).
Output: A sorted list L

```
c ← (a + b)/2                                   // Integer division: get mid point
L_1 ← merge-sort(L, a, c)
L_2 ← merge-sort(L, (c + 1), b)
return merge(L_1, L_2)
```

Algorithm 15 `merge-sort`(L) - Improved pseudocode

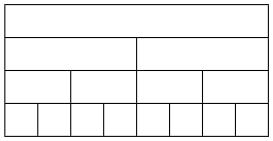
Input: A list of integers L.
Output: S sorted list L.

```
if |L| ≤ 1 then
    return L
Let L_1 be the first half of L (i.e. L[1], ..., L[n/2])
Let L_2 be the second half of L (i.e. L[n/2 + 1], ..., L[n])
L_1 ← merge-sort(L_1)
L_2 ← merge-sort(L_2)
return merge(L_1, L_2)
```

- *Question b goes here*

  Each time merge-sort is called recursively, the length of the argument halves. We can represent the various calls to merge-sort as a table in which the calls at level i are represented by the i-th row:

  Table 1: Number of merges per call
  The width of the table is n.



  Thus, the total amount of work done by `merge` at each level is $O(n)$.
  The total number of levels is $\log n$, which is roughly equals to the number of times n can be divided by 2 before reaching 1.
  Therefore, the total work for a list L of length n is $O(n \log n)$.

- *Question c goes here*

  Draw example trees for $h = 0$, $h = 1$ and $h = 2$ and count the number of leaves (respectively 1, 2, 4, etc...).
  Thus, for a tree of height h, there are $m = 2^h$ leaves. Hence, $h = \log_2 m$

- *Question d goes here*

  $n! = n(n-1) \times ... \times 2 \times 1$
  In this product, at least $\frac{n}{2}$ of the numbers are at least $\frac{n}{2}$ (and none of them is 0).

- *Question e goes here*

  Suppose A is a sorting algorithm based on comparisons. Consider the comparisons made in the various runs of A on a list $a_1, ..., a_n$ (a diagram should be drawn).
  Suppose, *without loss of generality*, all comparisons are binary. The time taken for A to run is the height of the tree (and the number of comparisons that you do).
  The worst case running time is equal to the length of longest branch.

Number of leaves = Number of permutations of $1, ..., n = n!$
In a full binary with height h we have $m = 2^h$ leaves. Certainly then, in our tree:

$$\begin{aligned} h &\geq \log(n!) \\ &\geq \log\left(\frac{n}{2}^{\frac{n}{2}}\right) \text{ (part d)} \\ \log\left(\frac{n}{2}^{\frac{n}{2}}\right) &= \frac{n}{2}\log\frac{n}{2} \\ &= \frac{n}{2}\log n - \frac{n}{2} \\ &= \frac{1}{2}\log n \end{aligned}$$

But $\Omega(n \log n)$ is a set of functions eventually bounded below by some multiple of $n \log n$. Thus, this function is $\Omega(n \log n)$ - at least as fast as some multiples of $n \log n$.

6.2 Euclid's algorithm

- *Let a and b be positive integers with $a \geq b$. Show that, if $r = a \mod b$, then*

$$\gcd(a, b) = \gcd(b, r).$$

Here, $\gcd(m, n)$ is the greatest common divisor (or highest common factor) of positive integers m and n.

$r = a \mod b$ by assumption.
Thus, $a = q \times b + r$
and $r = q \times b - a$.
Thus, any number b and r divides (b and) a, and any number dividing b and a divides (b and) r.
Thus, a, b and b, r have the same common factors, and have the same highest common factor.

- *Using pseudocode, give Euclid's algorithm for computing the highest common factor of two positive integers, and explain why it is correct.*

Algorithm 16 `hcf`(a, b)

Input: Non-negative integers a, b (Assume $0 \neq a \geq b$).
Output: $\gcd(a, b)$
  if $b = 0$ **then**
    return a
  return `hcf`$(b, amodb)$

- *What is the asymptotic running time of the algorithm you gave in Part 1b as a function of the size of (= number of bits in) the input? Justify your answer.*

Suppose the values of a in the recursive calls to $hcf(a, b)$ are $a, a_1, a_2, a_3, ...$ - cleary, $a_1 > a_2 > a_3...$ (the first inequality follows by assumption, the remainder cannot be bigger than the quotient since $r = a \mod b$ where $b = a_h$, $r = a_h + 1$).
Certainly, the algorithm terminates after l steps.
We claim $a_h + 2 \leq a_h/2$ for all $h \leq l - 2$.
If $a_h + 1 \leq (a_h)/2$, then the result follows because $a_h + 2 \leq a_h + 1$. Otherwise, i.e. if $a_h + 1 > (a_h)/2$, then since $a_h + 2 = a_h \mod a_h + 1$ (from algorithm), it results in $a_h + 2 \leq a_h + 1 \leq a_h/2$. This establishes the claim that the number of times a can be halved before we reach 0 is $\lceil \log a \rceil$, so the total work is logarithmic in a. Hence, it is linear in the size of a.