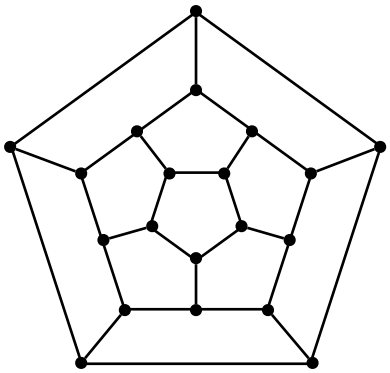
Assignment 4: Fun with Languages (30%)

Choose **one** of the following three topics.

Requirements for the assignment are specified at the end of each topic.

1. HUNT THE WUMPUS

Early computer games were often text-based. Hunt the Wumpus was created by Gregory Yob in the 1970s. After seeing a number of monster hunting games in a 1973 issue of *People's Computer Company* newsletter, Yob became annoyed that they all used a 10×10 grid. Many of these games were based on the concept of moving through a series of connected rooms around some fictional theme. In Hunt the Wumpus, the player explores a network of twenty numbered caves, each with three exits forming a *dodecahedron*, and a complex series of interactions (see the squashed dodecahedron below). Within this environment there are perils and adversaries. Two random rooms contain bottomless pits which result in instant death if the player stops into them. Two or more rooms are home to "super bats" who transport the player to another room at random. A fifth room contains the Wumpus, who light kill the player, or simply slink away through one of the three exits.



What makes this game interesting is that unlike modern games which are very visual, it is not possible to see the hazards directly - by the time you enter a room containing a hazard it is too late. Instead the player has to infer them from a distance. When one room away from pits, bats or the monster, the game provides messages: "I feel a draft", "Bats nearby", and "I smell a wumpus" respectively. The rumpus itself may be dealt with from a distance by firing a "crooked arrow" once you figure out which room it is hiding in.

Little is known about the elusive Wumpus, as all those who stumble upon it are immediately eaten. There does, however, seem to be a fairly universal consensus that it has suckers, which allow it to navigate the deadly Pits scattered throughout its labyrinthine home. The only hope the Agent has of killing the Wumpus without being eaten is to fire an arrow at it from afar, taking the Wumpus by surprise.

TASK

Given the original code written in BASIC, translate the program into a dialect of Fortran ≥ F95. Include the following:

- Provide the user with a visual made from ASCII characters to display the room, and its labeled connected rooms.
- Include a means of displaying messages such as "I smell a wumpus", i.e. something interesting.
- Make sure the code is modular and contains at least three (3) subprograms.

REFLECTION REPORT

Include a succinct 1 page summary (12-point, single-spaced).

Briefly describe your experiences translating the program from BASIC to Fortran. This could include a list of the benefits/limitations of Fortran over BASIC.

DELIVERABLES

The submission should consist of the following items:

- The reflection report (PDF).
- The code (well documented and styled appropriately of course): **`wumpus.f95`**
- Both the code and the reflection report should be submitted as a ZIP, TAR, or GZIP file.

FURTHER READING

- Ahl, D.H., Green, B., The Best of Creative Computing, (1976)

2. CALCULATING e TO MANY DIGITS

There are many algorithms for calculating numbers like π and *e*. Normally *e* is calculated using the infinite series:

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots$$

This series has a fast convergence, and is easy to calculate, however the problem is that computers are generally unable to accommodate more than 10-20 significant digits. For generating an accurate value for *e*, one has to turn to an alternative algorithm. One such algorithm is provided by Sale[1].

The code for this algorithm, from the original paper, is provided in Algol-60.

TASK

You are given a procedure, `ecalculation()` to perform the *e* algorithm written in Algol-60.

1. Translate the Algol-60 procedure to Fortran, Ada, C, and Python.

2. For each language add a wrapper main program which calls the procedure. The program file should be called `calce.X`, where `X` is the relevant language extension. The main program should also:
   2.1. Prompt the user for the number of significant digits to calculate.
   2.2. Prompt the user for the name of the file in which to store the value of *e* calculated.
   2.3. Call the subprogram, `keepe()`.

3. For each language, create a subprogram, `keepe()`, which saves the value of *e* calculated in an ASCII file. It takes as input the calculated value of `e`, and the filename specified by the user.

TESTING

You can test the accuracy of the programs, by comparing the value of *e* in the user specified file against a file containing the first 800 decimal places of *e* (using **cmp**).

REFLECTION REPORT

---

[1] Sale, A.H.J., "The calculation of e to many significant digits", The Computer Journal, 11(2), pp.229-230 (1968)

Include a succinct 1 page summary (12-point, single-spaced).

Describe your experiences implementing the algorithm in each of the languages: Fortran, Ada, C and Python. This could include a list of the benefits/limitations of each of the languages.

DELIVERABLES

The submission should consist of the following items:

- The reflection report (PDF).
- The code (well documented and styled appropriately of course): **`calce.f95, calce.adb, calce.c, calce.py`**
- Both the code and the reflection report should be submitted as a ZIP, TAR, or GZIP file.

THE *e* PROCEDURE IN Algol-60

```
procedure ecalculation (n, d);
value n;
integer n;
integer array d;
comment This procedure for calculating the transcendental
  number e to n correct decimal places uses only integer
  arithmetic, except for estimating the required series
  length. The digits of the result are placed in the array
  d, the array element d[0] containing entier(e), and the
  subsequent elements the following digits. These digits
  are individually calculated and may be printed one-by-
  one within the for statement labelled 'sweep'.
  begin integer m;
  real test;
  m := 4;
  test := (n + 1) × 2·30258509;
  loop: m := m + 1;
  if m × (ln(m) − 1·0) + 0·5 × ln(6·2831852 × m)
    ⩽ test then go to loop;
  begin integer i,j,carry,temp;
    integer array coef [2 : m];
    for j := 2 step 1 until m do coef [j] := 1;
    d [0] := 2;
    sweep: for i := 1 step 1 until n do begin
      carry := 0;
      for j := m step −1 until 2 do begin
        temp := coef [j] × 10 + carry;
        carry := temp ÷ j;
        coef[j]:= temp − carry × j
      end of digit generation;
      d [i] := carry
    end having calculated n digits
  end deleting declarations
end of ecalculation;
```

3. MERGESORT USING LINKED LIST

Mergesort is a divid-and-conquer algorithm for sorting lists. It was invented by Donald Knuth in 1945. It is fundamentally very simple:

1. Divide the list down the middle.
2. Sort each half.
3. Merge the halves.

The version of mergesort provided here involves applying the algorithm to a dynamic list of unsorted numbers. It is implemented in Pascal. The program contains a number of different functions:

- `buildlist(x,n)`
  - Builds a linked list from the `n` elements in array `x`.
- `printlist`
  - Prints out the linked list.
- `mergesort(p)`
  - Divides a list in half, sorts recursively and merges.
- `divide(p,q)`
  - Takes the list to which `p` points, divides it in half, and returns with `q` pointing to the start of the second half.
- `merge(p,q)`
  - Merges the lists to which `p` and `q` point, returning a pointer to the merged list.

TASK

Given a program for Mergesort written in Pascal, perform the following tasks:

- Translate the program to **Fortran** (`mrgsort.f95`), and **Ada** (`mrgsort.adb`).
- For Fortran create a **module** to store the merge functions (`mergefunc.f95`).
- For Ada create a **package** to store the merge functions (`mergefunc.ads/.adb`).
- For each language, modify the code so that a the name of a file containing unsorted integers can be input by the user (a file will be provided for testing). Output the sorted integers to a file named `sortedX.txt` (where `X=F` or `A`, depending on whether the program is Fortran, or Ada respectively).
- Each language should be appropriately modular, including such subprograms for tasks such as reading the integers from file, writing the sorted list to file.
- Provide relevant error handling for files that do not exist, or invalid data.

THE USER INTERFACE

The user interface will be very simple. It will be terminal input and output. For example:

```
> ./a.out
 Name of file to be sorted >
 unsorted.txt
 Sorry, file does not exist.
 Name of file to be sorted >
 unsorted1.txt
 Sorted integers saved to file sortedF.txt.
>
```

REFLECTION REPORT

Include a succinct 1 page summary (12-point, single-spaced).

Briefly describe your experiences translating the program from Pascal to Ada and Fortran. This could include a list of the benefits/limitations of the languages.

DELIVERABLES

The submission should consist of the following items:

- The reflection report (PDF).
- The code (well documented and styled appropriately of course):
  `mrgsort.f95, mergefunc.f95`
  `mrgsort.adb, mergefunc.adb, mergefunc.ads`
- Both the code and the reflection report should be submitted as a ZIP, TAR, or GZIP file.

FURTHER READING

- Knuth, D. E. "Sorting by Merging.", in Section 5.2.4 in The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd ed. Reading, MA: Addison-Wesley, pp. 158-168, 1998.

THE *mergesort* PROGRAM IN Pascal

```
program mergesort(input, output);
type pointer = ^node;
     node = record
              data : integer;
              next : pointer
            end;
     numlist = array [1..1000] of integer;

var unsorted : numlist;
    list : pointer;
    n : integer;

{ Builds a list from an array of integers }
procedure buildlist(x: numlist; n: integer);
var p: pointer;
    i : integer;
begin
   list := nil;
   for i := 1 to n do
   begin
      new(p);
      p^.data := x[i];
      p^.next := list;
      list := p
   end;
end;

{ Reads a list of integers in from the standard input }
procedure readints(var x: numlist; var n: integer);
var i: integer;
begin
   writeln('n?');
   read(n);
   for i := 1 to n do
   begin
      read(x[i])
   end;
end;

{ Print a list }
procedure printlist;
var pt : pointer;
begin
   pt := list;
   while pt <> NIL do
   begin
      write(pt^.data,' ');
      pt := pt^.next
   end;
   writeln;
end;

{ Takes a list, p, divides it in half, and returns with p pointing to the
head of the first half, and q to the head of the second half. }
procedure divide(var p,q: pointer);
```

```
var r: pointer;
begin
   q := p;
   r := p^.next;
   r := r^.next;
   while r <> nil do
   begin
      r := r^.next;
      q := q^.next;
      if r <> nil then r := r^.next
   end;
   r := q^.next;
   q^.next := nil;
   q := r
end;

{ Merges two sorted lists into one. Requires both lists be non-empty. }
function merge(p,q: pointer): pointer;
var r: pointer;
begin
   if (p = nil) or (q = nil) then
      writeln('merge called with empty list');
   if p^.data <= q^.data then
   begin
      r := p;
      p := p^.next
   end
   else begin
      r := q;
   end
   else begin
      r := q;
      q := q^.next
   end;
   merge := r;
   while (p <> nil) and (q <> nil) do
      if p^.data <= q^.data then
      begin
         r^.next := p;
         r := p;
         p := p^.next
      end
      else begin
         r^.next := q;
         r := q;
         q := q^.next
      end;
   if p = nil then r^.next := q else r^.next := p
end;

{ Divides list in helf, sorts recursively, and merges. }
procedure mergesort(var p: pointer);
var q: pointer;
begin
   if p <> nil then if p^.next <> nil then
   begin
      divide(p,q);
      mergesort(p);
```

```
      mergesort(q);
      p := merge(p,q)
   end;
end;

begin
   readints(unsorted,n);
   buildlist(unsorted,n);
   printlist;
   mergesort(list);
   printlist;
end.
```