Choose Any TWO Tasks.

<u>Task A (45%): Interpolation Search</u> max 500 words + equations/figures/tables + code

Interpolation search (Peterson, 1957), also known as predictive search, is a search algorithm for sorted arrays that is ordinarily implemented by repeated application of the linear interpolation formula given in Eq. 1.

$$i = \left\lfloor (top - bottom)\left(\frac{k - V_{bottom}}{V_{top} - V_{bottom}}\right) + bottom \right\rfloor \qquad \text{Eq. 1}$$

where

V	is our array.
top	is the upper index of the array segment we are searching, initially $n-1$ or n.
$bottom$	is the lower index of the array segment we are searching, initially 0 or 1.
V_{bottom}	is the value in the array at index $bottom$.
V_{top}	is the value in the array at index top.
k	is the key of the item that we are seeking (i.e., the search target).
i	is the array index predicted to contain key k.

Following each prediction of the target key's location we either find the target key at V_i, or a portion of V is eliminated (including the predicted index, i, itself) and Eq. 1 is re-run on the remaining array segment.

As discussed in class, following the calculation of i, if we compare V_i to our target key, k, we observe a best case of 1 comparison (exact). This occurs when the first prediction falls immediately on the target key. We also discussed that if the predicted index i is repeatedly as bad as it possibly can be, because the values in the array do not approximate an arithmetic progression (A.P.), then the extreme worst case will be n comparisons (exact). This occurs when every prediction results in the elimination of just one element from the array. It was also stated, but not elaborated upon, that the average case performance of interpolation search is loglogarithmically related to n. In big-O notation, this leads to summary Table 1.

best	average	worst
$\mathbf{O(1)}$	$\mathrm{O(\log\log n)}$	$\mathrm{O(n)}$

Table 1. Time complexity of interpolation search.

Interpolation search performs badly when the array segment eliminated after a failed prediction is small, and the retained segment is large. For our purposes, we will define large as meaning more than ¾ of the remaining array.

Question:
How would performance be affected if we intercepted bad predictions (as defined above) and applied a binary search step instead? (i.e., ignore prediction i and examine the middle element of the remaining array segment and retain the segment that might contain the target). In the next iteration, a new prediction would be made (which could again potentially be rejected). Does this approach outperform regular interpolation search, or binary search?

You will consider this scenario both theoretically (i.e., adjusting and explaining big-O and exact time measures) and empirically using code. Your code will comprise original and modified interpolation search test functions, binary search, and a test harness program that will collect the number of comparisons performed for randomly generated sorted arrays of different lengths that either grow as an approximation of an A.P. or otherwise. Importantly, we will measure performance using the dominant unit of work, i.e., *number of comparisons*, and we will not measure execution time. This code can be written in any language (except Scratch!). If text-based, save results into a file for plotting a graph (e.g., in Excel) of work done in comparisons (y-axis) for each array length (x-axis) for each approach.

References:
Peterson, W. W. (1957). Addressing for Random-Access Storage. *IBM Journal of Research and Development,* **1**(2), 130–146. doi: 10.1147/rd.12.0130.

Task B (45%): Filial-heir Chains

max 500 words + equations/figures/tables

Filial-heir chains, also known as left-child-right-sibling (LCRS) trees and doubly chained trees (Sussenguth, 1963), are rooted binary trees that can be used to represent k-ary trees (also known as k-way trees) in which $k > 2$.

Using pen-and-paper experiments and some intuition, you are to propose a set of theorems that describe the structure of filial-heir chains in relation to their equivalent k-ary tree. In particular, present theorems that predict metrics of interest following forward and reverse Knuth transforms (i.e., the process of converting a k-ary tree into a filial-heir chain and vice versa).

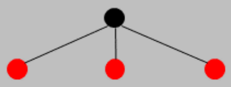
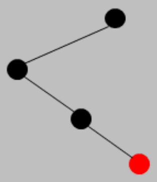
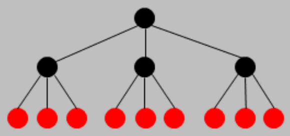
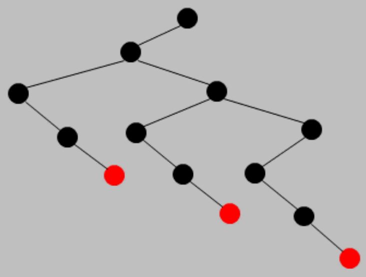
Metrics of interest include, but are not limited to: numbers of edges, nodes, leaf nodes, and interior nodes; tree height and balance, number of null pointers; maximum, minimum and average number of nodes in each level, and diameter (sometimes known as width). More marks are to be awarded for theorems that make less straightforward connections (although the formulas proposed should be as simple/direct as possible). An example of the type of theorem that could be proposed, and how your theorems should be presented, is given below:

Theorem 1.

If there are f_{3ary} leaf nodes in a complete 3-ary tree ($k = 3$), the number of leaf nodes in the equivalent filial-heir chain, f_{fht}, is given by:

$$f_{fht} = \left\lceil \frac{f_{3ary}}{k} \right\rceil$$

Demonstration:

	3-ary tree	filial-heir chain	
base case $h = 1$			$f_{fht} = \left\lceil \frac{f_{3ary}}{k} \right\rceil = \left\lceil \frac{1}{3} \right\rceil = 1$
$h = 2$			$f_{fht} = \left\lceil \frac{f_{3ary}}{k} \right\rceil = \left\lceil \frac{3}{3} \right\rceil = 1$
$h = 3$			$f_{fht} = \left\lceil \frac{f_{3ary}}{k} \right\rceil = \left\lceil \frac{9}{3} \right\rceil = 3$

Start with k-ary trees that are have the simplest structure (i.e., maximally degenerate and perfect), and if you are feeling ambitious, proceed to k-ary trees that are complete, balanced (at a specific, stated order) but not complete, and then trees that are neither perfect, complete, nor balanced.

Start with a particular value of k (e.g., $k = 3$), and then verify that your theorems hold for other values of k, or adjust them accordingly to make them generalisable. You are to use an inductive step to test your theorems (as illustrated above); test if the theorem holds when an independent variable is 1 (the base case), and then that it holds both when the independent variable has value x and $x + 1$. Avoid very small values of x, where possible.

Typeset your theorems in Microsoft Word using the equation editor (or LaTeX if you prefer) and present them as shown above. You should develop approximately 10, although as noted above, more marks will be awarded for theorems that are less obvious, especially if they can be applied more generally (e.g., using k as additional input).

References:

Sussenguth, E.H. (1963). Use of tree structures for processing files. *Communications of the ACM*, **6**(5), 272–279. doi:10.1145/366552.366600

Task C (45%): Partition Problem max 500 words + equations/figures/tables + code

In the partition problem, we ask whether we can divide a multiset (alternatively known as a bag or mset) S into two partitions (S_1 and S_2) that have equal sums (Eq. 2).

$$\sum_{x \in S_1} x = \sum_{y \in S_2} y \qquad \text{Eq. 2}$$

Note that it is not required that $|S_1| = |S_2|$; in the extreme case, there may be one element of S in S_1 and all other elements of S in S_2. Recall also that *multiset* implies that S (and consequently S_1 and S_2) can contain duplicates, unlike regular sets. For example, if $S = \{2,2,1,10,5,15,4,1\}$ then we can produce $S_1 = \{5,15\}$ and $S_2 = \{2,2,1,1,10,4\}$, both of which sum to 20 (i.e., the difference between the sums of S_1 and S_2 is 0).

This deceptively simple problem is computationally demanding. For a set of cardinality n there are 2^n possible subsets, giving exponential time complexity $O(2^n)$ if a brute force approach is used (i.e., trying all possibilities). It is traditionally framed as an NP-complete decision problem (i.e., producing a yes or no result, depending upon whether an equal-sum partition is or is not possible). We can also frame it as an optimisation problem that aims to minimise the absolute difference between the two sums. For instance, if $S = \{2,6,15,10\}$ then the optimal partition is $S_1 = \{2,15\}$ and $S_2 = \{6,10\}$, which have sums of 17 and 16 respectively, giving an absolute difference of 1. It is the optimisation variant of the partition problem that we shall consider here.

First, we will empirically and theoretically compare several simple approaches to see how well they minimise the absolute difference between the partitions they produce over a range of cardinalities of S. Our multiset S can be implemented as an array in any programming language of your choice (although choose wisely; e.g., OO will unnecessarily complicate matters). The array should be populated with randomly generated numbers in the range 1 to the $10 \times$ the current cardinality. We will examine the following cardinalities: 32, 64, 128, 256, 512, and 1024. Since our objective is to divide S to yield two partitions with equal sum, the *ideal* sum is equal to the sum of all values in S divided by 2. The approaches to be compared are:

A. Divide S into two equally-sized partitions (S_1 and S_2) by splitting the array in the middle.

B. Take even elements of S for S_1 and odd elements of S for S_2.

C. Add the first element of S to S_1 and the second element to S_2. Now iterate through the remaining elements, adding them to whichever partition currently has the smallest sum. This is sometimes called greedy partitioning.

D. Divide S array into two sub-arrays comprising values $\leq \bar{S}$ and values $> \bar{S}$. We will again use a greedy approach. First iterate through the array of larger values and allocate each to whichever partition has the smallest current sum. Next iterate through the array of smaller values and allocate those to whichever partition has the smallest current sum.

E. Sort S into ascending order and then take even elements for S_1 and odd numbered elements for S_2.

F. First sort S into descending order. Add the first element of S to S_1 and the second element to S_2. Now iterate through the remaining elements, adding them to whichever partition currently has the smallest sum.

For each approach A..F, provide a walkthrough example (similar to those above), state the time complexity in big-O notation, and provide a line graph with multiset cardinality (array length) on the x-axis and mean absolute partition difference on the y-axis. This can be created by exporting your results from <u>lots</u> of repetitions of your test program to plot in Excel, or using a language that supports graph plotting, like MATLAB or Python. Your graph should also contain error bars around each mean (e.g., 1 standard deviation) to show the dispersion of your samples around that mean. This will also help to confirm that you have a sufficient number of results to make reliable inferences.

Next, research, describe, implement, and test one additional approach (but not brute force), G., that works *better* than A..F, above. Each part (A..F) is worth 5%, and G is worth 15% of your mark.

References

Karmarkar, N. and Karp, R.M. (1982). The differencing method of set partitioning. Technical Report UCB/CSD 81/113, Computer Science Division, University of California, Berkeley.

Presentation Checklist (10% if and only if <u>all</u> are done):

Report Guidelines	**Tick Box**
I confirm that my write-ups for each of the two tasks I have chosen to submit are free from spelling, typographical, and grammatical errors (i.e., I have carefully proof-read my work, and have used the Word spelling and grammar check functions, or equivalent).	☐
I confirm that my work is written in the third person (i.e., doesn't use "I" or any other personal pronouns) in order to maintain scientific clarity and objectivity, and is as concise and factual as possible.	☐
I confirm that all figures, tables and equations have been numbered sequentially, starting from one (Fig. 1, Fig. 2, etc for figures), that there is an explanatory caption stating what the figure/table shows (see examples in this document), and that all figures/tables/equations are referred to in the text.	☐
I confirm that Harvard referencing has been used to clearly identify all third-party sources used in the creation of my submission, which are also listed at the rear of the document for each task.	☐
I confirm that I have remained below the word limit for each task (500 words), and understand that equations, figures, tables, code snippets and pseudocode are not counted as words.	☐
I confirm that I understand that I am to upload my files to Canvas by the deadline. I will upload a main document named to match my SID in PDF or DOCX format (e.g., `1234567.docx`), plus a ZIP file containing my code.	☐

Figure, Table and Equation Guidelines	**Tick Box**
I confirm that I have labelled the x and y axes of all line graphs, and that the lines appearing in these line graphs are clearly identified in the figure captions provided underneath each figure, and optionally also in a figure legend.	☐
I confirm that all equations have been typeset (e.g., using the Word Equation Editor or LaTeX) rather than being copied/pasted from elsewhere or written in regular type, and that if equations correspond to code/pseudocode fragments, then equation variable names match those used in my code.	☐

Code and Pseudocode Guidelines	**Tick Box**
I confirm all variable names make sense with respect to their roles in my code, and I have used camelCaps for multi-word variable names and UPPER CASE for constants.	☐
I confirm that my code is well commented, paying particular attention to lines of code that are more difficult to understand on casual inspection.	☐
I confirm that I have avoided single letter variable names wherever possible, except where these relate directly to widely accepted mathematical notation (e.g, in Task C, S, S_1, S_2, etc are acceptable).	☐
I confirm that my code has been properly indented so that nested logic (iteration, selection) is very obvious to the reader.	☐
I confirm that my code has been printed for inclusion in my report in a fixed-width-font such as Courier or Courier New to preserve the integrity of indentation and general readability.	☐
I confirm that my code is as short as possible (does not perform unnecessary work), and does not include unreachable or non-functioning lines of code, and that different units of code within a task are harmonised in terms of the way they are presented (variable names, function arguments, etc).	☐