Instructions

Your submission is graded (pass/fail) based on what you submit on learnit alone. Bear in mind that grading will be done based on the learning objectives—it is possible to pass even if your code/answers have some flaws. The CodeJudge is hopefully running during the exam and is meant only as a debugging aid and to help you meet the specification of the input and output.

We expect a `.pdf` file (including the ITU standard frontpage from Study Guide) containing your answers to the text questions, as well as all source code for your implementations. State the compiler and command line options of the compiler that you are using if there are any unusual options (for example vanilla Java and most code that compiles on CodeJudge probably do not need this—in short, make sure that we have enough information to run your code).

You are allowed to use any external source. If you do so, you have to give a reference. I.e.: You can copy paste small pieces of code from publicly available sources as much as you want as long as it is clear which part of your implementation (or text) is literally copied, or based on a copy. If we find that you used a source without referencing it, we will consider this a case of cheating. Before the deadline for handing in, you are not supposed to discuss this exam with anybody (by whatever medium) and you are not supposed to share your solutions or receive outside advice or solutions. If we find indications (in the check directly after the exam, while grading or in any other way) that you received such help, we will report it as a suspected case of cheating.

Please choose two out of three questions to be graded. You may simply submit two answers, or clearly mark which of your solutions you wish to be graded. If you submit three answers without marking which are to be graded, we will grade questions 1 and 2 by default.

1 Independent Set in Interval Graphs

In this problem, we look at intervals on the real line $[\ell, r]$. In particular, each interval corresponds to the set of numbers from ℓ to r. We say two intervals intersect if they have any points in common: for example, $[1, 4]$ and $[2, 5]$ intersect; so do $[1, 2]$ and $[2, 3]$.

Consider a set of n intervals S. Our goal is to find the maximum independent (i.e. non-intersecting) set of intervals.[1] See Figure 1.

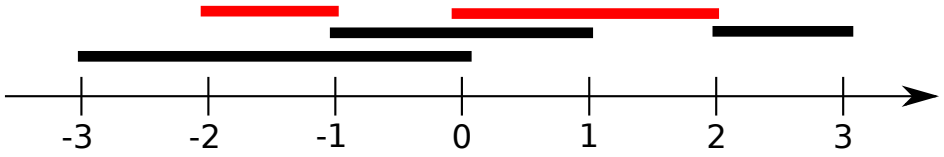

Figure 1: An instance with intervals $[-3, 0], [-2, -1], [-1, 1], [0, 2], [2, 3]$. Algorithm 1 gives the solution $[-2, -1], [0, 2]$, shown in red. This is an optimal solution—there is no set of three non-intersecting intervals.

This problem can be solved optimally by the following algorithm. This algorithm greedily adds intervals to our independent set one at a time, in order of earliest to latest right endpoint (i.e. earliest to latest r). After we add a given interval, we must remove all intervals it intersects with (as they can no longer be a part of the final solution).

Algorithm 1 Independent Set

Create two linked lists A_ℓ and A_r.
$A_\ell \leftarrow A$ sorted in increasing order of ℓ (the left endpoint).
$A_r \leftarrow A$ sorted in increasing order of r (the right endpoint).
$\mathcal{T} \leftarrow \emptyset$ ▷ This will store our independent set
while A_r is nonempty **do**
    Let $[\ell, r]$ be the first interval in A_r.
    Add $[\ell, r]$ to $\mathcal{T}$.
    Remove $[\ell, r]$ from A_r.
    Find $[\ell, r]$ in A_ℓ and remove it.
    Let $[\ell_2, r_2]$ be the first interval in A_ℓ.
    while $\ell_2 \leq r$ **do**
        Find $[\ell_2, r_2]$ in A_r and remove it.
        Remove $[\ell_2, r_2]$ from A_ℓ.
        Let $[\ell_2, r_2]$ be the first interval in A_ℓ.

1.1 Implement

Implement the above algorithm. We have provided CodeJudge exercises which you can use to test your implementation.

The input intervals are given in a text file. Each interval is given as two integers on a line separated by space. The expected output is formatted the same way, with intervals in sorted order by right endpoint. (The algorithm will generate the intervals in this order automatically.)

[1] By maximum, we mean the set with the largest number of non-intersecting intervals.

The algorithm takes two arguments: the name of the file containing the intervals, and the number of input intervals.

1.2 Questions to Discuss

(a) How fast is your implementation?
(b) Can you obtain $O(n)$ time after the initial sorting step? Give a brief explanation as to why.
(c) How do you ensure that we can jump between A_r and A_ℓ quickly (in Step 8 for example)?
(d) Why do we use $\ell_2 \leq r$ as the condition in the inner loop?
(e) How would you implement this if A_r and A_ℓ were priority queues? What operations would you need the priority queues to support?

2 Summing Triples of Integers

Given a list S of integers in a file, your task is to determine if there are three integers a, b, c (each of which is in S) such that $a + b = c$. It may be that two or three of a, b, and c are the same—in other words, we can reuse members of S.

For example, the set $\{12, 15, 27, 53, 54, 76, 81, 103, 228\}$ is a "yes" instance because $27+54 = 81$. Similarly, $\{6, 8, 9, 12, 19, 31, 33\}$ and $\{0, 71, 73, 89\}$ are both "yes" instances because $6 + 6 = 12$ and $0 + 0 = 0$ respectively. However, $\{2, 27, 103, 431\}$ is a "no" instance as it contains no such triple.

S is given as a list of positive integers, one per line. The program should take two arguments: the first is the name of the file containing the set, and the second is the number of items in the set. The number of items in the set is guaranteed to be a power of two.[2] Example arguments would be `128numbersYES.txt 128`

We have given exercises on CodeJudge to allow you to verify your implementation.

Algorithm 2 Find Triple

```
procedure FindTriple(S, n)
    numBuckets ← n/16.
    shift ← log2(numBuckets).
    seed ← random odd positive number.
    Create numBuckets buckets.
    for each element x of S do
        h ← Hash(x, seed, shift)
        Store x in Bucket h.
    for each pair of buckets i, j do                ▷ This includes the case where i = j..
        k1 ← (i + j) % numBuckets.
        if NaiveCompare(Bucket i, Bucket j, Bucket k1) = 1 then return 1
        k2 ← (i + j + 1) % numBuckets.
        if NaiveCompare(Bucket i, Bucket j, Bucket k2) = 1 then return 1
    return 0
procedure Hash(x, seed, shift)
    return (x * seed) ≫ (64 − shift)
procedure NaiveCompare(Bucket A, Bucket B, Bucket C)
    for Each element a in A do
        for Each element b in B do
            for Each element c in C do
                if a + b = c then return 1
    return 0
```

Here is an implementation of `Hash`. It has an extra step to help it run on most 64-bit platforms (no matter how right shift works). If you are using 32-bit numbers, change the 64 to a 32.

```
long hash(long input, long seed, long shift) {
    return  ((input * seed) >> (64 - shift))  &  ((2 << (shift-1)) - 1);
}
```

[2] Otherwise `shift` would not be an integer—this is easy to handle but we avoid it for simplicity.

Some comments about the above algorithm:

- Even though there is randomness, if implemented correctly this algorithm always gives the right answer.
- Comments about `shift`: Instead of $\log_2$, one can use ρ from the ANF implementation that gives the number of leading zeros in `numBuckets`. Example values: if $n = 1024$, `numBuckets` should be 64, and `shift` should be 6. If $n = 128$, `numBuckets` should be 8 and `shift` should be 3.
- In the HASH procedure, $*$ denotes multiplication, and $\gg$ denotes *unsigned* right shift.[3] HASH should always return an integer between 0 and `numBuckets` $- 1$.
- One can significantly speed up procedure NAIVECOMPARE (it runs in $O(n^3)$ as described, but $O(n^2)$ is easily possible by hashing C). Feel free to make this improvement if you want, but it is not required for the question.
- For some intuition, this algorithm works because (working modulo `numBuckets`) if $a + b = c$, then $h(a) + h(b) = h(c)$ or $h(a) + h(b) = h(c) + 1$, where h is the hash procedure above.

Implement

(a) Implement the above algorithm. (Note that there are simpler algorithms that would solve this problem; for example one could compare all triples using the brute force algorithm alone; in fact an optimized brute force approach can run very quickly. However, this question asks you to specifically use the bucketing-based idea in Algorithm 2.)

Questions to Discuss

(a) What data structure did you use to store the buckets? What are some advantages and disadvantages of this approach?

(b) In the pseudocode above there is a constant 16. Why do we have this constant (in other words, what high-level concept from class is achieved by making this constant higher than 1)?[4] When porting this code to a new platform (i.e. a new machine), what aspects of the new machine might cause us to reconsider this choice, and test to see if a new constant leads to better performance?

[3] This unsigned aspect is why there is an extra & in the provided code.

[4] Hint: changing this constant will change the average size of the buckets. This question is asking why we would want a moderate bucket size.

3 Two Kinds of Bubblesort

This question deals with two kinds of bubble sort. Up-Shifting Bubble Sort is standard bubble sort: we iterate over elements, swapping each element with the next element if it is smaller. Alternating Bubble Sort alternates between two kinds of iterations: in the first it swaps each element with the next, as usual. In the second, it does the reverse, swapping each element with the previous if it is larger.

Algorithm 3 Up-Shifting Bubble Sort

```
procedure BubbleSort(A, size)
    flag ← true
    while flag do
        flag ← false
        for x from 0 to size − 2 do
            if A[x] > A[x + 1] then
                Swap(A[x], A[x + 1])
                flag ← true
```

Algorithm 4 Alternating Bubble Sort

```
procedure BubbleSort(A, size)
    flag ← true
    while flag do
        flag ← false
        for x from 0 to size − 2 do
            if A[x] > A[x + 1] then
                Swap(A[x], A[x + 1])
                flag ← true
        if flag then
            flag ← false
            for x from size − 1 to 1 do
                if A[x] < A[x − 1] then
                    Swap(A[x], A[x − 1])
                    flag ← true
```

This question tests experimental design (note that we will also ask you to perform the actual experiments in this question). As such, please be specific about your experimental design, giving a complete design as described in Chapter 2 of McGeoch. In cases where two experiments are very similar, you may shorten your response to explain how they differ—just be sure that for each experiment, someone reading your report can recover all the details of your experimental design.

When performing experiments, be sure to describe the trend you see in the experiments, and try to provide an explanation for the trend. We do not have a specific required input format (nor a CodeJudge test); feel free to use whatever format is most convenient to perform your experiments.

3.1 Simple Experiment

We begin with a simple test of Up-Shifting Bubble Sort.

Design and carry out an experiment to compare the performance of Up-Shifting Bubble Sort on random data vs already-sorted data. (You do not need to repeat the experiment using Alternating Bubble Sort.) Include a p-value analysis of your results. (Only one p-value analysis is necessary.)

3.2 k inversions

Consider an input generated using k inversions. In particular, consider an input generated by the following process:

- Start with a sorted array A.
- Repeat k times: Randomly pick a pair (i, j) and swap $A[i]$ and $A[j]$.

Design and carry out an experiment to compare Up-Shifting Bubble Sort and Alternating Bubble Sort on the above input for various values of k.

3.3 Nearly-Sorted Data

Consider a list of integers A which is *nearly sorted*. In particular, an element at index i in A is guaranteed to be at position at least $i - k$ and at most $i + k$ in sorted order. See Figure 2.

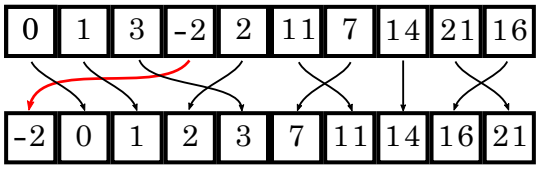

Figure 2: An example of nearly-sorted data, with $k = 3$. $A[3] = -2$ is distance 3 away from its final position (in red); all other points are at distance less than 3.

Design and carry out an experiment to test Up-Shifting Bubblesort and Alternating Bubblesort on nearly-sorted data. In particular, your experiment should generate nearly sorted inputs. Your test does not need to be perfect, but it should be reasonable. In particular, elements may not be exactly uniformly distributed between $i - k$ and $i + k$, but it should run reasonably efficiently and generate some reasonably interesting cases.

(a) How do you generate nearly-sorted input?
(b) What are some advantages and disadvantages of your approach?
(c) Perform an experiment where you fix n and vary k. With the results of this experiment, what do you think the asymptotic runtime of Up-Shifting Bubble Sort is on this input? What about Alternating Bubble Sort?