Term Test 1 NWEN303 2021

Total 100 points

Duration: 2 hours

There are 3 questions, with sub questions.

Answer in the white space after each sub question. Each sub question gives you a full page to answer, but you are not expected to fill up all the space. Many questions have elegant short answers.

Write your name on the top right corner of all the pages.

Question 1

A web server busy with many concurrent requests hosts the following code to sort a list of products when the price is in a given range. The code is wrote in a traditional sequential manner as shown below.

```
-----------------
class Location { int shippingCostTo(Location l){/*..*/} }
class Product{
  int price; String description; Location location;
  Product(int price, String description, Location location){
    this.price=price; this.description=description; this.location=location;
    }
  public String toString() {return "Product("+/*..*/+")";}
  }
/*..*/
static List<Product> priceFrom(int low, int high, List<Product> products){
  List<Product> res=new ArrayList<>();
  for(Product p: products){
    if(p.price<low){continue;}
    if(p.price>high){continue;}
    res.add(p);
  }
  Collections.sort(res,(p1,p2)->p1.price-p2.price);
  return Collections.unmodifiableList(res);
}
-----------------
```

Q1a [20Marks]: Rewrite the code of '`priceFrom`' using parallel streams in the best way.

```
static List<Product> priceFrom(int low, int high, List<Product> products){
  return products.parallelStream()
    .filter(p->p.price>=low)
    .filter(p->p.price<=high)
    .sorted((p1,p2)->p1.price-p2.price)
    .collect(Collectors.toUnmodifiableList());
  }
```

Q1b [15Marks]: Even with the best version of the code, we can observe that the performance of the web server under load is now decreased. Explain why.

Programming is all about multiple layers of code calling each other.
In the context of a web server there are two very clear layers:

- TOP: receive all the requests and dispatch them to an algorithm to manage them
- REQUEST: manage an individual request

All webservers are very good at parallelize the TOP part, so that many requests can be executed in parallel.
If for example there are 8 cores and in a given moment there are 500 requests, 8 requests can be handled in parallel.
Handling those 8 requests would use all the computational capacity of those cores.
Thus, if in the lower layer of computation we try to parallize further, we do not have access to any more computational potential: the hardware is all busy already.
It is going to be slower because of the intrinsic overhead in trying to set up a parallel computation.

As we discussed in the first lecture: We should not try to parallelize code that runs under a parallel environment, like a web server.

Question 2

Products can be in storage all around the world, so to find the best deal we need to consider shipping. However, computing the shipping cost is very slow, so we want to cache the shipping cost to a fixed destination, to avoid recomputing it over and over again.

The algorithm shown below is correct, but it is atrociously ugly, verbose and it is still quite slow. As you can see from the comment, the author wanted the second part to run in parallel, but they could not even figure out how to do so.

```
-----------------
static int cost(Product p,Location destination){//takes long time to compute
shipping
  return p.price+p.location.shippingCostTo(destination);
  }
static Product bestDeal0( List<Product> products, Location destination ) {
  //allocate space to store cached costs
  ArrayList<Integer> costs = new ArrayList<Integer>(products.size());
  //there is a bestDeal only on non empty lists
  if( !products.isEmpty() ) {
    //allocate the space to store workers
    List<CompletableFuture<Integer>> futs
      = new ArrayList<CompletableFuture<Integer>>(products.size());
    for( Product p : products ) {
      //initialize all the parallel work
      futs.add(CompletableFuture.supplyAsync(()->cost(p,destination)));
      }
    for( CompletableFuture<Integer> fut : futs ) {
      //wait for the work to complete and fill the cache
      costs.add(fut.join());
      }
    Product res = null;//ok, never read
    int minCost = Integer.MAX_VALUE; //can not be bigger than that!
    for( int i=0; i<products.size(); i++ ) {//can this be parallelized too?
      Product current = products.get(i);//custom min implementation
      int currentCost = costs.get(i);//using indexes and the cache
      if( currentCost<minCost ) {
        res=current;
        minCost=currentCost;
        }
      }
    return res;
    }
  else {
    throw new AssertionError("No product available");
    }
  }
-----------------
```

Q2a [16Marks]:
Identify and describe at least 4 problems in the code below.
Include at least one causing performance loss and one causing verbosity

-BestDeal0 is not a good name, bestDeal would be better?

-We can add javadocs instead of comments all over the place

-Costs is declared before the if: if the if fails, we just throw error without using costs, so we have initialized a new array list for nothing. costs should as minimum go inside the if.

-The if-else should be swapped: in this way we avoid a layer of indentation for most of the code

-The pattern 'allocate a list of futures', for all the tasks make a future, wait for all the futures is much better done with parallel streams.

-min is a very common computaition, probably we can use a library call for it.

Q2b [19Marks]:
Rewrite this algorithm in an efficient, compact and elegant way.

```
record ProductCost(Product product, int cost){}

static Product bestDeal( List<Product> products, Location destination ) {
  if( products.isEmpty() ){throw new AssertionError("No product available");}
  return products.parallelStream()
    .map(p->new ProductCost(p,cost(p,destination)))
    .min((p1,p2)->p1.cost()-p2.cost()).get().product();
  }
```

Question 3:

The code below attempts to execute in parallel the code of ‘computeA’ and ‘computeB’ and then combine the result.
It has 3 mistakes. For each of those describe why it is a problem and what effect it has on a program using ‘question3’.

```
-----------------
static int computeA(int input) {/*..*/}
static String computeB(int input) {/*..*/}
static ExecutorService pool=Executors.newFixedThreadPool(/*..*/);
static int question3() throws Throwable{
  Future<Integer> a=pool.submit(()->computeA(1));
  Future<String> b=pool.submit(()->computeB(2));
  pool.shutdown();
  return a.get()+b.get().length();
  }
-----------------
```

Q3a [10Marks]: describe the first mistake and the effect it has on a program using ‘`question3`’.

The code '`throws Throwable`' instead of the most specific exceptions InterruptedException and ExecutionException.
This DOES NOT cause any problem during the execution of the method, but make life a nightmare for the user of such method.

Q3b [10Marks]: describe the second mistake and the effect it has on a program using ‘`question3`’.

To split in 2 using futures, you just need to use a future for the FIRST task, and you can execute the second one directly, without the need of a second submission to the pool.

Q3c [10Marks]: describe the third mistake and the effect it has on a program using '`question3`'.

pool.shutdown() should NOT be called in the method.
It would work fine for the FIRST execution, but any attempt of using this method multiple time would fail.
Overall, pools should only be closed in the main or in some computation with top level control.