COMP2402A Midterm Exam — Fall 2012 — 1h20m

Please answer all questions on the provided Scantron sheet. Select only a single answer for each question. In case multiple answers are correct, select a single answer that best, or most precisely, answers the question.

The question sheet has 11 pages; it **may not** be taken after the exam.

In all instances, `n` denotes the number of elements currently stored in whichever data structure the question is discussing.

1. Which of the Java Collections Framework interfaces would be the most useful if we want to store a collection of Social Insurance Numbers (SINs) and the only operation we ever want to do is check if a particular SIN is part of the collection?

   (a) `Collection`

   (b) `Set`

   (c) `SortedSet`

   (d) `Map`

   (e) `SortedMap`

2. What, if in addition, we want to store information (e.g., Name and Date of Birth) associated with each SIN?

   (a) `Collection`

   (b) `Set`

   (c) `SortedSet`

   (d) `Map`

   (e) `SortedMap`

3. What if, in addition to the first two requirements, we also want to be able to output a list of all records, sorted numerically by SIN?

   (a) `Collection`

   (b) `Set`

   (c) `SortedSet`

   (d) `Map`

   (e) `SortedMap`

4. A `Bag` is like a `Set` except that equal elements can be stored more than once. Which of the following is best suited to implement a `Bag<T>`?

   (a) `Set<T>`

   (b) `Map<T,Integer>`

   (c) `Map<T,List<T>>`

   (d) `SortedSet<T>`

   (e) Either (b) or (c) depending on what behaviour we want if we add two elements that are equal but not identical.

5. The running time of the methods `get(i)` and `add(i,x)` for an `ArrayList` are

   (a) $O(1+\mathtt{i})$ and $O(1+\mathtt{i})$, respectively

   (b) $O(1)$ and $O(1)$, respectively

(c) $O(1+\mathtt{i})$ and $O(1+\mathtt{n}-\mathtt{i})$, respectively
(d) $O(1)$ and $O(1+\mathtt{n}-\mathtt{i})$, respectively
(e) $O(1)$ and $O(1+\mathtt{i})$, respectively

6. The running time of the methods `get(i)` and `add(i,x)` for a `LinkedList` are

   (a) $O(1+\mathtt{n}-\mathtt{i})$ and $O(1)$, respectively
   (b) $O(1+\mathtt{i})$ and $O(1+\mathtt{i})$, respectively
   (c) $O(1+\mathtt{i})$ and $O(1+\mathtt{n}-\mathtt{i})$, respectively
   (d) $O(1)$ and $O(1+\mathtt{n}-\mathtt{i})$, respectively
   (e) $O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$ and $O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$, respectively

7.
```
public static void frontGets(List<Integer> l, int n) {
  for (int i = 0; i < n; i++) {
    l.get(0);
  }
}
```
   The above method is

   (a) much faster when `l` is a `LinkedList`
   (b) much faster when `l` is an `ArrayList`
   (c) about the same speed independent of whether `l` is an `ArrayList` or a `LinkedList`

8.
```
public static void randomGets(List<Integer> l, int n) {
  Random gen = new Random();
  for (int i = 0; i < n; i++) {
    l.get(gen.nextInt(l.size()));
  }
}
```
   The above method is

   (a) much faster when `l` is a `LinkedList`
   (b) much faster when `l` is an `ArrayList`
   (c) about the same speed independent of whether `l` is an `ArrayList` or a `LinkedList`

9.
```
public static void removeAtBack(List<Integer> l, int n) {
  for (int i = 0; i < n; i++) {
    l.remove(l.size()-1);
  }
}
```
   The above method is

   (a) much faster when `l` is a `LinkedList`
   (b) much faster when `l` is an `ArrayList`
   (c) about the same speed independent of whether `l` is an `ArrayList` or a `LinkedList`

10.
```
public static void removeFront(List<Integer> l, int n) {
  for (int i = 0; i < n; i++) {
    l.remove(0);
  }
}
```
   The above method is

(a) much faster when `l` is a `LinkedList`

(b) much faster when `l` is an `ArrayList`

(c) about the same speed independent of whether `l` is an `ArrayList` or a `LinkedList`

11.
```
public static void insertInMiddle(List<Integer> l, int n) {
  for (int i = 0; i < n; i++) {
    l.add(new Integer(i));
  }
  for (int i = 0; i < n; i++) {
    l.add(n/2+i, new Integer(i));
  }
}
```
The above method is

(a) much faster when `l` is a `LinkedList`

(b) much faster when `l` is an `ArrayList`

(c) about the same speed independent of whether `l` is an `ArrayList` or a `LinkedList`

12.
```
public static void insertInMiddle2(List<Integer> l, int n) {
  for (int i = 0; i < n; i++) {
    l.add(new Integer(i));
  }
  ListIterator<Integer> li = l.listIterator(n/2);
  for (int i = 0; i < n; i++) {
    li.add(new Integer(i));
  }
}
```
The above method is

(a) much faster when `l` is a `LinkedList`

(b) much faster when `l` is an `ArrayList`

(c) about the same speed independent of whether `l` is an `ArrayList` or a `LinkedList`

13. Recall that an `ArrayStack` stores `n` elements in a backing array `a` at locations `a[0]`,...,`a[n-1]`:

```
public class ArrayStack<T> extends AbstractList<T> {
  T[] a;
  int n;
  ...
}
```

Also recall that, immediately after the backing array `a` is resized by `resize()` it has `a.length` = 2`n`.

When adding an element, the `ArrayStack` grows the backing array `a` if it is full, i.e, if `a.length` = `n`.

If are currently about to grow the backing array `a`, what can you say about the number of `add()` and `remove()` operations (as a function of the current value of `n`) since the last time the `ArrayStack` was resized?

(a) At least `n`/2 `remove()` operations have occurred since then

(b) At least 2`n`/3 `add()` operations have occurred since then

(c) At least `n`/2 `add()` operations have occurred since then

(d) At least $2\texttt{n}/3$ `remove()` operations have occurred since then

(e) We can not bound either the number of `add()` nor `remove()` operations

14. Recall that we shrink the backing array `a` when $3\texttt{n} < \texttt{a.length}$. If we are currently about to shrink the backing array `a`, what can you say about the number of `add()` and `remove()` operations since the last time the `ArrayStack` was resized?

    (a) At least $\texttt{n}/2$ `remove()` operations have occurred since then

    (b) At least $\texttt{n}/2$ `add()` operations have occurred since then

    (c) At least $2\texttt{n}/3$ `remove()` operations have occurred since then

    (d) At least $2\texttt{n}/3$ `add()` operations have occurred since then

    (e) We can not bound either the number of `add()` nor `remove()` operations

15. From the previous two questions, what can you conclude about the total number of elements copied by `resize()` if we start with an empty `ArrayStack` and perform `m` `add()` and `remove` operations.

    (a) At most $2\texttt{m}$ elements are copied by `resize()`

    (b) At most `m` elements are copied by `resize()`

    (c) At least `m` elements are copied by `resize()`

    (d) We can not bound the number of elements copied by `resize()`

    (e) At least $2\texttt{m}$ elements are copied by `resize()`

16. Suppose that the `ArrayStack` implementation were modified the following ways:

    - `resize()` resizes the backing array so that its size is $3\texttt{n}$
    - `remove()` only calls `resize()` when $6\texttt{n} < \texttt{a.length}$.

    If we are currently about to resize the backing array `a`, what can you say about the number of `add()` and `remove()` operations since the last time the `ArrayStack` was resized?

    (a) At least `n` `add()` or `remove()` operations have occurred since then

    (b) At least $\texttt{n}/2$ `add()` operations or `n` `remove()` operations have occurred since then

    (c) At least $\texttt{n}/2$ `add()` or `remove()` operations have occurred since then

    (d) At least $2\texttt{n}/3$ `add()` or `n` `remove()` operations have occurred since then

    (e) At least `n` `add()` operations or $\texttt{n}/2$ `remove()` operations have occurred since then

17. Recall that an `ArrayDeque` stores `n` elements at locations `a[j]`, `a[(j+1)%a.length]`,...,`a[(j+n-1)%a.length]`:

```
public class ArrayDeque<T> extends AbstractList<T> {
  T[] a;
  int j;
  int n;
  ...
}
```

    What is the amortized running time of the `add(i,x)` and `remove(i)` operations?

    (a) $O(1 + \texttt{i})$

    (b) $O(1 + \texttt{n} - \texttt{i})$

    (c) $O(1 + |\texttt{i} - \texttt{n}/2|)$

(d) $O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$

(e) $O(1+\min\{\mathtt{i}-\mathtt{n},\mathtt{n}-\mathtt{i}\})$

18. If $\mathtt{m} = 2^{10}$ then the binary representations of `m` and $\mathtt{m}-1$ are

    (a) `01111111111` and `10000000000`, respectively

    (b) `10000000000` and `09999999999`, respectively

    (c) `10000000000` and `01111111111`, respectively

    (d) `10000000000` and `11111111111`, respectively

    (e) `10101010001` and `00101010111`, respectively

19. From the previous question, if the binary representation of `x` is `0001110001110001110001010101010101`, then the binary representation of `x%m` is

    (a) `0001110001110001110001010101010101`

    (b) `0001110001110001110001010000000000`

    (c) `0000000000000000000111000111000111`

    (d) `0000000000000000000000000101010101`

    (e) `0000000000000000000000000000000000`

20. Recall that a `DualArrayDeque` implements the `List` interface using two `ArrayStacks`:

```
public class DualArrayDeque<T> extends AbstractList<T> {
  ArrayStack<T> front;
  ArrayStack<T> back;
  ...
}
```

    In order to implement `get(i)` we need to get it from the `ArrayStack`, `front` or `back`. We can express this as

    (a) `front.get(i)`

    (b) `front.get(front.size()-i-1)`

    (c) `back.get(i-front.size())`

    (d) Either (a) or (c) depending on the value of `i` and `front.size()`

    (e) Either (b) or (c) depending on the value of `i` and `front.size()`

21. When a `DualArrayDeque` is rebalanced (by the `rebalance()` method) it is because $|\mathtt{front.size()} - \mathtt{back.size()}| > \mathtt{n}/2$. After rebalancing, $|\mathtt{front.size()} - \mathtt{back.size()}| \le 1$. What can you conclude about the number of operations since the last rebalancing occured?

    (a) At least $\mathtt{n}/2-1$ `add()` operations have occurred since then

    (b) At least $\mathtt{n}/2-1$ `remove()` operations have occurred since then

    (c) At least $\mathtt{n}/2-1$ `add()` and/or `remove()` operations have occurred since then

    (d) At least $2\mathtt{n}/3-1$ `remove()` operations have occurred since then

    (e) At least $2\mathtt{n}/3-1$ `add()` operations have occurred since then

22. Recall that a `RootishArrayStack` stores a list in a sequence of arrays (blocks) of sizes 1, 2, 3, 4,. . . .

```
public class RootishArrayStack<T> extends AbstractList<T> {
  List<T[]> blocks;
  int n;
  ...
}
```

If a `RootishArrayStack` has 11 blocks (so `b.size()` = 10), then how many elements can it store?

(a) 90
(b) 132
(c) 66
(d) 55
(e) none of the above

23. In a `RootishArrayStack`, a call to `get(25)` will return

(a) `blocks.get(25)[0]`
(b) `blocks.get(4)[6]`
(c) `blocks.get(6)[9]`
(d) `blocks.get(6)[4]`
(e) `blocks.get(9)[6]`

24. Recall our implementation of a singly-linked list (`SLList`):

```
protected class Node {
  T x;
  Node next;
}
public class SLList<T> extends AbstractList<T> {
  Node head;
  Node tail;
  int n;
  ...
}
```

Consider how to implement a `Queue` as an `SLList`. When we enqueue (`add(x)`) an element, where does it go? When we dequeue (`remove()`) an element, where does it come from?

(a) We enqueue (`add(x)`) at the `head` and we dequeue (`remove()`) at the `head`
(b) We enqueue (`add(x)`) at the `head` and we dequeue (`remove()`) at the `tail`
(c) We enqueue (`add(x)`) at the `tail` and we dequeue (`remove()`) at the `tail`
(d) We enqueue (`add(x)`) at the `tail` and we dequeue (`remove()`) at the `head`
(e) None of the above

25. Consider how to implement a `Stack` as an `SLList`. When we push an element where does it go? When we pop an element where does it come from?

(a) We push at the `tail` and we pop at the `head`
(b) We push at the `tail` and we pop at the `tail`

(c) We push at the `head` and we pop at the `tail`
(d) We push at the `head` and we pop at the `head`
(e) None of the above

26. Using the best method you can think of, how quickly can we find the ith node in an SLList?

(a) in $O(1 + \min\{\mathtt{i}, \mathtt{n} \cdot (\mathtt{n} - \mathtt{i} - 1)\})$ time
(b) in $O(1 + \mathtt{i})$ time
(c) in $O(1 + \mathtt{n} - \mathtt{i})$ time
(d) in $O(1 + \mathtt{n} - \mathtt{i})$ time
(e) in $O(1 + \min\{\mathtt{i}, \mathtt{n} - \mathtt{i}\})$ time

27. Recall our implementation of a doubly-linked list (`DLList`):

```
protected class Node {
  Node next, prev;
  T x;
}
public class DLList<T> extends AbstractSequentialList<T> {
  protected Node dummy;
  protected int n;
  ...
}
```

Explain the role of the `dummy` node. In particular, if our list is non-empty, then what are `dummy.next` and `dummy.prev`?

(a) `dummy.next` is the last node in the list and `dummy.prev` is the first node in the list
(b) `dummy.next` and `dummy.prev` are both the first node in the list
(c) `dummy.next` and `dummy.prev` are both the last node in the list
(d) `dummy.next` is the first node in the list and `dummy.prev` is the last node in the list
(e) None of the above is true

28. Consider the correctness of the following two methods that add a node `u` before the node `p` in a `DLList`.

```
protected Node add(Node u, Node p) {
  u.next = p;
  u.next.prev = u;
  u.prev = p.prev;
  u.prev.next = u;
  n++;
  return u;
}
protected Node add(Node u, Node p) {
  u.next = p;
  u.prev = p.prev;
  u.next.prev = u;
  u.prev.next = u;
  n++;
  return u;
}
```

(a) The first method is correct
(b) The second method is correct
(c) Neither method is correct
(d) Both methods are correct
(e) Both (c) and (d)

29. What is the running-time of `add(i,x)` and `remove(i)` in a `DLList`?

(a) $O(1+\mathtt{i})$ and $O(1+\mathtt{i})$, respectively
(b) $O(1)$ and $O(1+\mathtt{n}-\mathtt{i})$, respectively
(c) $O(1+\mathtt{n}-\mathtt{i})$ and $O(1)$, respectively
(d) $O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$ and $O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$, respectively
(e) $O(1)$ and $O(1+\mathtt{n}-\mathtt{i})$, respectively

30. Recall that, in our implementation of a space-efficient linked-list (`SEList`), the list is stored as a sequence of blocks, each represented by a bounded `ArrayDeque` capable of storing up to $\mathtt{b}+1$ elements.

To ensure that an `SEList` does not use too much memory we maintain the invariant that

(a) Every block, except possibly the last block, contains exactly `b` elements.
(b) Every block, except possibly the last block, contains at most $\mathtt{b}+1$ elements.
(c) Every block, except possibly the last block, contains at least $\mathtt{b}+1$ elements.
(d) Every block, except possibly the last block, contains at most $\mathtt{b}-1$ elements.
(e) Every block, except possibly the last block, contains at least $\mathtt{b}-1$ elements.

31. An `SEList` ensures that the total amount of wasted space (space not used to directly store elements) is most

(a) $O(\mathtt{n})$
(b) $O(\mathtt{n}-\mathtt{i})$
(c) $O(\mathtt{n}/\mathtt{b})$
(d) $O(\mathtt{b})$
(e) $O(\mathtt{n}/\mathtt{b}+\mathtt{b})$

32. The running time of the `get(i)` method in an `SEList` is

(a) $O(1+\mathtt{i})$
(b) $O(1+\mathtt{n})$
(c) $O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$
(d) $O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\}/\mathtt{b})$
(e) $O(\mathtt{b})$

33. If we place `n` distinct elements into a hash table of size `m` using a good hash function, how many elements do we expect to find in each table position?

(a) $O(\mathtt{n}/\mathtt{m})$
(b) $O(\mathtt{m}/\mathtt{n})$
(c) $O(\mathtt{n})$
(d) $O(\mathtt{m})$

(e) $O(\mathtt{n} \cdot \mathtt{m})$

34. Recall the multiplicative hash function `hash(x) = (x.hashCode() * z) >>> w-d`, where `w` is the number of bits in an integer. How large is the table that is used with this hash function? (In other words, what is the *range* of this hash function?)

    (a) $\{0, \ldots, 2^{\mathtt{w}-\mathtt{d}}\}$

    (b) $\{0, \ldots, 2^{\mathtt{d}}\}$

    (c) $\{0, \ldots, 2^{\mathtt{d}} - 1\}$

    (d) $\{0, \ldots, 2^{\mathtt{w}} - 1\}$

    (e) $\{0, \ldots, 2^{\mathtt{w}-\mathtt{d}} - 1\}$

35. In more standard mathematical notation, the above hash function can be written as (here div denotes integer division without any remainder)

    (a) $\mathrm{hash}(\mathtt{x}) = ((\mathtt{x.hashCode()} \cdot \mathtt{z}) \bmod 2^{\mathtt{w}-\mathtt{d}}) \operatorname{div} 2^{\mathtt{d}}$

    (b) $\mathrm{hash}(\mathtt{x}) = (\mathtt{x.hashCode()} \cdot \mathtt{z}) \operatorname{div} 2^{\mathtt{w}-\mathtt{d}}$

    (c) $\mathrm{hash}(\mathtt{x}) = ((\mathtt{x.hashCode()} \cdot \mathtt{z}) \bmod 2^{\mathtt{w}}) \operatorname{div} 2^{\mathtt{w}-\mathtt{d}}$

    (d) $\mathrm{hash}(\mathtt{x}) = ((\mathtt{x.hashCode()} \cdot \mathtt{z}) \bmod 2^{\mathtt{w}}) \operatorname{div} 2^{\mathtt{d}}$

    (e) $\mathrm{hash}(\mathtt{x}) = ((\mathtt{x.hashCode()} \cdot \mathtt{z}) \bmod 2^{\mathtt{w}-\mathtt{d}}) \operatorname{div} 2^{\mathtt{w}-\mathtt{d}}$

36. Consider the following implementation of a `hashCode()` method that uses the bitwise exclusive-or (^) operator

```
public class Point2D {
  Double x, y;
  ...
  public int hashCode() {
    return x.hashCode() ^ y.hashCode();
  }
}
```

    Which of the following statements are true about two instances `p` and `q` of a `Point2D`?

    (a) $\mathtt{p.hashCode()} = 0$ if $\mathtt{p.x} = \mathtt{p.y}$

    (b) $\mathtt{p.hashCode()} \neq \mathtt{q.hashCode()}$ if $\mathtt{p.x} \neq \mathtt{q.y}$ or $\mathtt{p.y} \neq \mathtt{q.x}$

    (c) $\mathtt{p.hashCode()} = \mathtt{q.hashCode()}$ if $\mathtt{p.x} = \mathtt{q.y}$ and $\mathtt{p.y} = \mathtt{q.x}$

    (d) Both (a) and (b) are true

    (e) Both (a) and (c) are true

37. Consider the following implementation of a `hashCode()` method

```
public class Point2D {
  Double x, y;
  ...
  public int hashCode() {
    return x.hashCode() + y.hashCode();
  }
}
```

Which of the following statements are true about two instances `p` and `q` of a `Point2D`?

(a) `p.hashCode()` $=0$ if `p.x` $=$ `p.y`

(b) `p.hashCode()` $\neq$ `q.hashCode()` if `p.x` $\neq$ `q.y` or `p.y` $\neq$ `q.x`

(c) `p.hashCode()` $=$ `q.hashCode()` if `p.x` $=$ `q.y` and `p.y` $=$ `q.x`

(d) Both (a) and (b) are true

(e) Both (a) and (c) are true

38. A software efficiency expert has an expert idea about how to speed up hashing of arrays of integers. His idea is to randomly choose a value $\mathtt{b} \in \{0, 1\}$ and only use the hash codes of the even (in the case $\mathtt{b} = 0$) or odd (in the case $\mathtt{b} = 1$) array locations.

    Which of the following statements is true about this expert method:

    (a) We can easily find `n` distinct arrays all of which have the same hash code

    (b) We can easily find `n` distinct arrays, and `n`/2 of them will have the same hash code

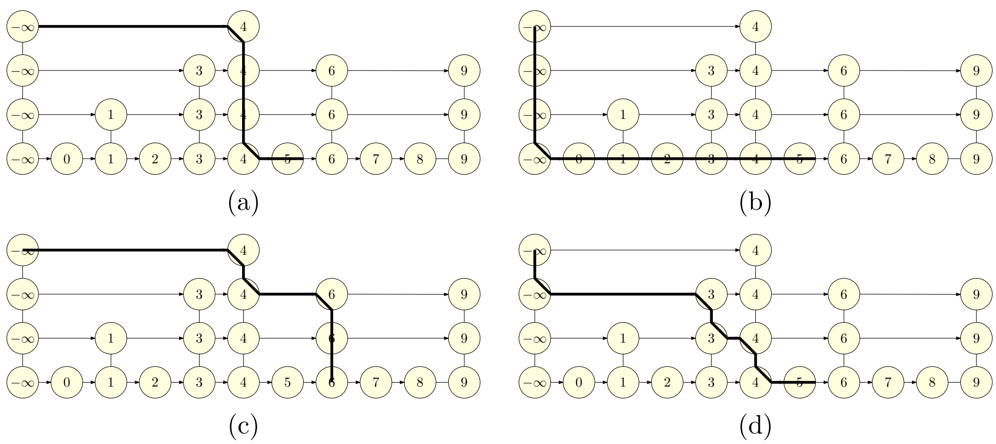
    (c) We can easily find two distinct arrays that have the same hash code

    (d) We can easily find `n` distinct arrays, and they will have only 2 distinct hash codes

    (e) None of the above

39. Recall that a skiplist stores elements in a sequence of smaller and smaller lists $L_0, \ldots, L_k$. L_i is obtained from L_{i-1} by tossing a coin for each element in L_{i-1} and including the element in L_i if that coin comes up heads.

    Which of the following pictures illustrates the search path for 6 in the skiplist?



(a) (b) (c) (d)

40. Tossing a coin and counting how many times it comes up heads before the first tail is related to which of the following quantities in a skiplist?

    (a) The total size of the skiplist

    (b) The number of steps the search path takes at a particular level

    (c) The number of lists a particular element x takes part in

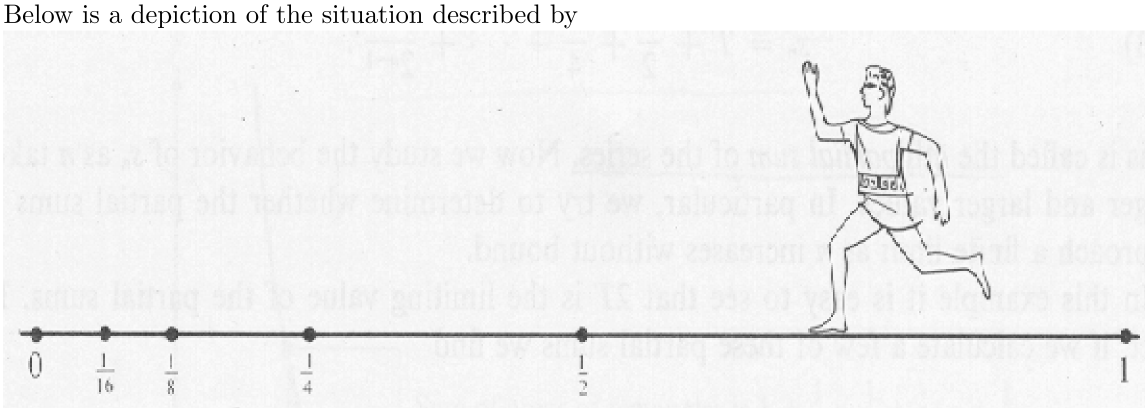
    (d) The total length of the search path

    (e) Both (b) and (c)

41. If the list L_0 contains n values, what is the expected number of elements in the list L_i?

(a) $2i$
(b) $i/2$
(c) 2^i
(d) $n/2^i$
(e) n^2

42. Below is a depiction of the situation described by



(a) Alan Turing
(b) Carl Friedrich Gauss
(c) Zeno of Elea
(d) Alfred E. Neumann
(e) Charles Babbage