Tentamensuppgifter

1.

```
public int sum(int n) {
	if (n == 1) {
		return 1;
	}
	else {
		return n + sum(n-1);
	}
}
```

Anropas med 5 – Anropas med 4 – Anropas med 3 – Anropas med 2 -
Anropas med 1 – Returnar 1 – Returnar hela vägen längs anropskedjan.

Rekursionsdjup 4/5 med lika många records på systemstacken. Innehåller informationen från föregående calls.

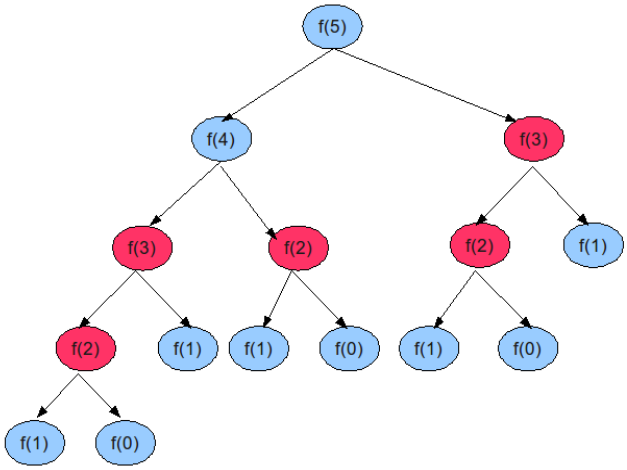
Bättre lösning: Använd formeln Summa = n(n+1)/2, alternativt en iterativ lösning.

Summan 1 + 2 + 3 + … + n = n(n+1)/2.
Steg1: Visa att formeln stämmer för ett bestämt fall, t.ex. n=1.
1(1+1)/2 = 2/2 = 1 – OK!
Steg2: Antag att formeln stämmer för ett godtyckligt fall, n.
Steg3: Visa att formeln stämmer för nästa tal, n+1.
Om vi har 1+2+3+…+n+(n+1) = (1+2+3+…+ n)+(n+1) = n(n+1)/2 + (n+1) = (n+1)(1+n/2) = (n+1)(n+2)/2
Därmed stämmer formeln även för näsföljande tal, och om den stämmer för 1, stämmer den för 2, osv.

2.

```
public int fibonacci(int n) {
	if (n <= 1) {
		return n;
	}
	else {
		return fibonacci(n-1) + fibonacci(n-2);
	}
}
```

Ineffektiv för stora n pga antalet anrop som behövs.
Den räknar dessutom ut samma sak flera gånger.




```
public static int fibonacci(int n) {
	int lower = 0;
	int higher = 1;
	int temp;
	for (int i = 0; i < n-1; i++) {
		temp = lower + higher;
		lower = higher;
		higher = temp;
	}
	return higher;
}
```

Den iterativa metoden är effektivare då den tidskrävande operationen (loopen) endast utförs n gånger.

3.

```
public void move(int n, int from, int to, int via) {
            if (n == 1) {
                  System.out.println("Move disk from pole " + from +
                  " to pole " + to);
            }
            else {
            move(n - 1, from, via, to);
            move(1, from, to, via);
            move(n - 1, via, to, from);
            }
      }
```

Körning med 3 diskar, starttornet: 1, måltornet: 3, hjälptornet: 2

```
Move disk from pole 1 to pole 3
Move disk from pole 1 to pole 2
Move disk from pole 3 to pole 2
Move disk from pole 1 to pole 3
Move disk from pole 2 to pole 1
Move disk from pole 2 to pole 3
Move disk from pole 1 to pole 3
```

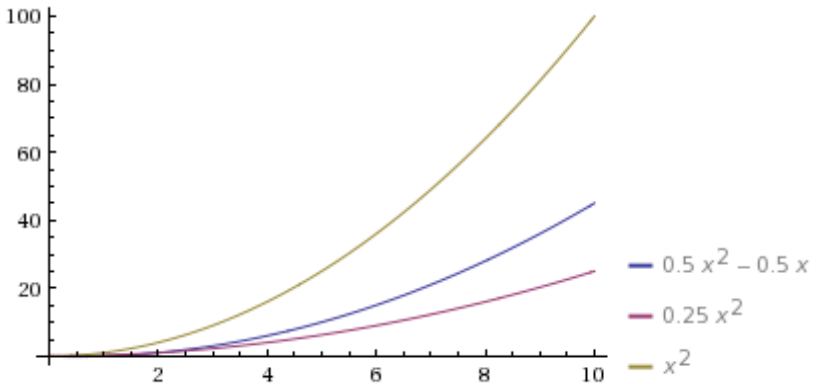
4.

1: Θ(n^2), 2: Θ(n^3), 3: Θ(n), 4: Θ(nlogn), 5: Θ(3^n), 6: Θ(n!)

5.

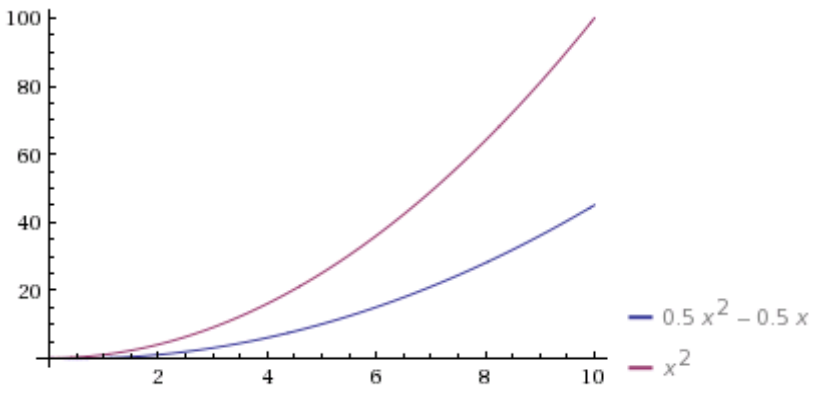
Θ(f(n)) Definition: Komplexitetsfunktionen tillhör mängden om den kan inneslutas i ett intervall av
cf(n) <= g(n) <= df(n) för n >= N
g(n) = 0.5n^2-0.5n tillhör Θ(n^2) då vid c=0.25 och d=1 gäller olikheten för n >= 2.
Detta kan också bevisas via lim g(n)/f(n) = konstant då n går mot oändligheten.
lim 0.5n^2-0.5n / n^2 = 0.5 (L'Hospital)

Alla komplexitetsfunktioner som inte har n^2 som dominerande faktor för stora n tillhör inte Θ(n^2).
t.ex. n^3 tillhör inte Θ(n^2). Lim n^3/n^2 kommer gå mot oändligheten när n går mot oändligheten.
Det går inte heller att begränsa funktionen cn^2 <= n^3 <= dn^2 för något c, d eller n.

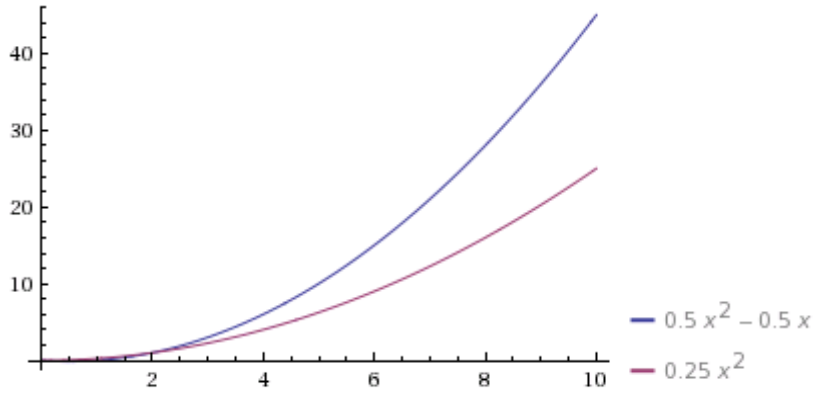


6.

O(f(n)) Definition: Komplexitetsfunktionen tillhör mängden om den uppifrån kan begränsas av
g(n) <= cf(n) för n >= N
g(n) = 0.5n^2-0.5n tillhör O(n^2) eftersom den kan begränsas av 0.5n^2-0.5n <= n^2 för n >= 0.
n^3 tillhör inte O(n^2) eftersom det inte går att begränsa n^3 <= cn^2 för något värde på c.



7.
Ω(f(n)) Definition:
Komplexitetsfunktionen tillhör mängden om den nedifrån kan begränsas av
cf(n) <= g(n) för n >= N
g(n) = 0.5n^2-0.5n tillhör Ω(n^2) eftersom den kan begränsas av
0.25n^2 <= 0.5n^2-0.5n för n >= 2.
g(n) = n tillhör inte Ω(n^2) eftersom det inte går att begränsa cn^2 <= n för något värde på c.



8.
```
public static int[]
sort(int[] nums) {
      for (int i = 0; i < nums.length-1; i++) {
            int smallestIndex = i;
            for (int k = i+1; k < nums.length; k++) {
                  if (nums[k] < nums[smallestIndex]) {
                        smallestIndex = k;
                  }
            }
            int temp = nums[i];
            nums[i] = nums[smallestIndex];
            nums[smallestIndex] = temp;
      }
      return nums;
}
```

Den största tidsåtgången går åt vid jämförelsen i den inre for-loopen. Den utförs först n-1 gånger, sen n-2 gånger, n-2 gånger osv. Summan blir alltså n(n-1)/2 jämförelser.
Algoritmens tidskomplexitet blir av storleksgrad Θ(n^2).

9.
```
public int[] insertSort(int[] nums){
      for(int i = 1; i < nums.length; i++){
            int value = nums[i];
      int j = i - 1;
      while(j >= 0 && nums[j] > value){
                  nums[j + 1] = nums[j];
                  j = j - 1;
      }
      nums[j + 1] = value;
      }
      return nums;
}
```
Algoritmen tar ut det värde som ska sorteras (börjar med det näst första talet), och går utmed arrayen bakåt tills ett tal som är lägre påträffas, då är talets plats innan det talet.
I värsta fall är arrayen i motsatt ordning, alltså måste man flytta elementen 1 plats, 2 platser, … n-2 platser, n-1 platser. Alltså har den i värsta fall samma tidskomplexitet som selectionsort, n(n-1)/2. Men i genomsnitt behövs hälften av elementen flyttas, alltså blir tidskomplexiteten i ett average case n(n-1)/4.

11.
Tidskomplexiteten: Båda algoritmerna är av storleksgraden Θ(nlogn) så de har liknande prestanda. Dock så har quicksort en något mindre tidskomplexitet i sitt average-case. Quicksort har ett värre worst-case än mergesort där tidskomplexiteten är av storleksgraden Θ(n^2) medan mergesort endast har Θ(n) i värsta fall.

Minneskomplexiteten: Merge behöver en extra array som är av samma storlek som har samma storlek som input-arrayen. Dessutom läggs ett antal recalls på stacken, maximalt logn, som försummas för stora n. Alltså blir minneskomplexiteten av storleksgrad Θ(n). Quicksort lägger i värsta fall n-1 recalls på systemstacken men i ett average case läggs endast logn recalls. Alltså blir minneskomplexiteten av storleksgrad Θ(logn).

12.

```
public static int binarySearch(int[] nums, int key, int min, int max) {
	if (max < min) {
		return -1;
	}
	else {
		int mid = (min + max) / 2;
		if (nums[mid] > key) {
			return binarySearch(nums, key, min, mid);
		}
		else if (nums[mid] < key) {
			return binarySearch(nums, key, mid+1, max);
		}
		else {
			return mid;
		}

	}
}
```

Tidskomplexiteten för binärsökning i värsta fall utgörs när det sökta elementet inte finns i arrayen. Alltså kommer algoritmen anropas tills man har ett intervall på endast ett element. Komplexiteten blir då logn + 1.

13.

LIFOQueue – I labuppgifterna (Lab 2)

15.

```
public boolean balance(String in) {
	ArrayDeque<Character> q = new ArrayDeque<Character>();
	for (int i = 0; i < in.length(); i++) {
		if (in.charAt(i) == '(') {
			q.addLast('(');
		}
		else if (in.charAt(i) == ')') {
			try {
				q.removeLast();
			}
			catch (NoSuchElementException e) {
				return false;
			}
		}
	}
	if (q.isEmpty()) {
		return true;
	}
	return false;
}
```

16.

```
public int postfix(String in) {
	ArrayDeque<Integer> q = new ArrayDeque<Integer>();
	String[] components = in.split(" ");
	for (int i = 0; i < components.length; i++) {
		if (components[i].equals("+")) {
			q.addLast(q.removeLast() + q.removeLast());
		}
		else if (components[i].equals("-")) {
			int temp = q.removeLast();
			q.addLast(q.removeLast() - temp);
		}
		else if (components[i].equals("*")) {
			q.addLast(q.removeLast() * q.removeLast());
		}
		else if (components[i].equals("/")) {
			int temp = q.removeLast();
			q.addLast(q.removeLast() / temp);
		}
		else {
			q.addLast(Integer.parseInt(components[i]));
		}
	}
	return q.removeLast();
}
```

20.

ParentIndex = (ChildIndex – 1) / 2

21.

En heap med n element har logn + 1 nivåer avrundat med heltalsräkning, alltså nedåt. Logaritmfunktionen har basen 2.

I värsta fall måste insättning och borttagningsoperationerna ordna om heapen så att man måste göra en jämförelse i varje nivå. Alltså blir tidskomplexiteten logn.

23.

```
public boolean contains(E element) {
	for (int i = 0; i < elements.length; i++) {
		int compValue = comparator.compare(element, elements[i]);
		if (compValue == 0) {
			return true;
		}
	}
	return false;
}
public boolean contains(E element) {
	Node currentNode = firstNode;
	while (currentNode != null) {
		int compValue = comparator.compare(element, currentNode.element);
		if (compValue == 0) {
			return true;
		}
		else {
			currentNode = currentNode.nextNode;
		}
	}
	return false;
}
```

24.

```
public void remove(E element) {
	for (int i = 0; i < elements.length; i++) {
		int compValue = comparator.compare(element, elements[i]);
		if (compValue == 0) {
			elements[i] = elements[lastIndex];
			elements[lastIndex] = null;
			break;
		}
	}
}
public void remove(E element) {
	Node currentNode = firstNode;
	Node parentNode = null;
	while (currentNode != null) {
		int compValue = comparator.compare(element, currentNode.element);
		if (compValue == 0) {
			if (parentNode == null) {
				firstNode = currentNode.nextNode;
			}
			else {
				parentNode.nextNode = currentNode.nextNode;
			}
		}
		else {
			parentNode = currentNode;
			currentNode = currentNode.nextNode;
		}
	}
}
```

26.

```
public void add(int index, E element){
	if ((index < 0) || (index > size())) {
		throw new IndexOutOfBoundsException("Index " + index +
							" felaktigt.");
	}
	if (lastIndex == elements.length - 1) {
		enlarge();
	}
	for (int i = lastIndex+1; i > index; i--) {
		elements[i] = elements[i - 1];
	}
	elements[index] = element;
	lastIndex++;
}
public void add(int index, E element){
	if ((index < 0) || (index > size())) {
		throw new IndexOutOfBoundsException("Index " + index +
							" felaktigt.");
	}
	Node newNode = new Node(element);
	Node currentNode = firstNode;
	for (int i = 0; i < index; i++) {
		currentNode = currentNode.nextNode;
	}
	newNode.nextNode = currentNode.nextNode;
	currentNode.nextNode = newNode;
	lastIndex++;
}
```

27

```
public void remove(int index){
	if ((index < 0) || (index > size())) {
		throw new IndexOutOfBoundsException("Index " + index +
							" felaktigt.");
	}
	for (int i = index; i > lastIndex; i++) {
		elements[i] = elements[i + 1];
	}
	elements[lastIndex] = null;
	lastIndex--;
}

public void remove(int index){
	if ((index < 0) || (index > size())) {
		throw new IndexOutOfBoundsException("Index " + index +
							" felaktigt.");
	}
	Node currentNode = firstNode;
	Node parentNode = null;
	for (int i = 0; i < index; i++) {
		parentNode = currentNode;
		currentNode = currentNode.nextNode;
	}
	if (parentNode == null) {
		firstNode = currentNode.nextNode;
	}
	else {
		parentNode.nextNode = currentNode.nextNode;
	}
}
```

28.

En backtrackingalgoritm undviker vissa av de möjligheter som inte leder till en lösning medan en brutefore-algoritm provar alla möjliga kombinationer tills den hittar en som passar. I fallet med damerna försöker algoritmen inte att placera flera damer på samma rad då det inte leder till någon lösning.

30.

```
public void print(Node node) {
	if (node != null) {
		print(node.leftNode);
		System.out.print(node.element + " ");
		print(node.rightNode);
	}
}
```

31.

```
private int size(Node node) {
      int    numberOfElements = 0;
      if (node != null) {
          numberOfElements = 1 + size(node.leftNode) + size(node.rightNode);
      }
      return numberOfElements;
}

public int size() {
      int numberOfElements = 0;
      if (root != null) {
            Deque<Node> queue = new ArrayDeque<Node>();
            Node currentNode = null;
            queue.addLast(root);
            while (!queue.isEmpty()) {
                  currentNode = queue.removeFirst();
                  numberOfElements++;
                  if (currentNode.leftNode != null) {
                        queue.addLast(currentNode.leftNode);
                  }
                  if (currentNode.rightNode != null) {
                        queue.addLast(currentNode.rightNode);
                  }
            }
      }
      return numberOfElements;
}
```

32.

```
public E get(E element, Node currentNode){
      int compare = element.compareTo(currentNode.element);
      if (compare == 0) {
            return element;
      }
      else if (compare == -1) {
            return get(element, currentNode.left);
      }
      else {
            return get(element, currentNode.right);
      }
}
```

33.

```
public Node insert(E element, Node currentNode){
      int compare = element.compareTo(currentNode.element);
      if (currentNode == null) {
            currentNode = new Node(element);
      }
      else {
            if (compare <= 0) {
                  currentNode.leftNode = insert(currentNode.leftNode, element)
            }
            else {
                  currentNode.rightNode = insert(currentNode.rightNode, element)
            }
      }
}
```