1) **a)**

Recurrente betrekking van algoritme:

$$T(n) = \begin{cases} 1 & als\ n{=}1 \\ T(n{-}1)^n & als\ n{>}1 \end{cases}$$

$T(1) = 1$

$T(2) = T(2-1)^2 = 2$

$T(3) = T(3-1)^3 = 6$

$T(4) = T(4-1)^4 = 24$

$T(n) = n!$

```
permutationlist[list.size!];
permutate('1..n', '', permutationlist)

func permutate(list, prefix, permlist) {
    if(prefix.length == list.size){
        permlist.add(prefix);
    }else{
        for(int i = 0; i < list.size; i++) {
            character c = list[i]
            list.remove(c);
            permutate(list, prefix + list[i], permlist);
            list.add(c);
        }
    }
}
```

Begin met alle permutaties van het kleinste getal. Zodra alle permutaties beginnend met het kleinste getal zijn gebeurd dan moeten alle permutaties van het volgende getal gedaan worden, totdat alle n getallen uit de lijst geweest zijn om te permuteren.

b) Complexiteit:

Big-Oh:

Er moet gelden:

$(\exists c,\ n_0 \in \mathbb{N})(\forall n > n_0)(g(n) \le cf(n))$

waarbij:

$g(n) = n!$
stel:
$c = 1$
dan geldt:
$n! \leq 1f(n) \Leftrightarrow n \leq f(n)$
hieruit kunnen we constateren dat:
$n! \in O(n!)$

Big-Omega:
Er moet gelden ofwel:
$g \notin o(f)$
ofwel:
$(\exists \varepsilon > 0)(\exists^{\infty} n_i)(\frac{g(n_i)}{f(n_i)} > \varepsilon)$
gebruik regel 1, hiervoor moet gelden dat:
$(\forall \varepsilon > 0)(\exists n_0 \in \mathbb{N})(\forall n > n_0)(g(n) < \varepsilon f(n))$
niet kan.
waarbij:
$g(n) = n!$
stel:
$\varepsilon = 1$
hieruit volgt dat:
$(\forall n > n_0)(g(n) < \varepsilon f(n))$
niet geldt.
Dus dan:
$n! \notin o(n!)$
En kunnen we constateren dat:
$n! \in \Omega(n!)$

Little-Theta:
er moet gelden:
$g \in O(f) \cap \Omega(f)$
Dit is net bewezen, dus:
$n! \in \theta(n!)$

c)

```
permutation = '';
permutate('1..n', '', 0)

func permutate(list, prefix, found) {
	if(prefix.length == list.size){
		permutation = prefix;
		return found++;
	}else{
		for(int i = 0; i < list.size; i++) {
			character c = list[i]
			list.remove(c);
			found = permutate(list, prefix + list[i], found);
			if(found == m)
				return found;
			list.add(c);
		}
	}
}
```

d) Dit algoritme heeft altijd m-stappen nodig om bij de juiste permutatie te komen $\theta(m)$. Als worst-case scenario is dit n!-stappen $O(n!)$. Dit gebeurt als de m-de volgorde die gevraagd wordt, de laatste permutatie is. Best-case heeft het algoritme maar 1 stap nodig $\Omega(1)$, dan staat de m-de volgorde op de eerste plaats.

2)

a) $O(n^2)$

Er moet gelden:

$(\exists c,\ n_0 \in \mathbb{N})(\forall n > n_0)(g(n) \le cf(n))$

waarbij:

$f(n) = n^2$

stel:

$g(n) = an^2 + bn + d$

dan geldt voor:

$c = a + b + d$

en

$n_0 = 0$

dat
$(\forall n > n_0)(g(n) \leq cf(n))$
geldt.
dus:
$an^2 + bn + d \in O(n^2)$

een algoritme met deze complexiteit is bijvoorbeeld bubble sort. Bij deze methode moet bij elke stap twee getallen moeten worden vergeleken totdat het grootste gevonden getal achteraan overblijft. Dit moet daarna herhaald worden voor de n-1 overgebleven mogelijkheden. Dit levert worst-case n^2 stappen op. Dus geldt hiervoor $O(n^2)$.

b) $\Omega(n)$
Er moet gelden ofwel:
$g \notin o(f)$
ofwel:
$(\exists \varepsilon > 0)(\exists^\infty n_i)(\frac{g(n_i)}{f(n_i)} > \varepsilon)$
gebruik regel 1, hiervoor moet gelden dat:
$(\forall \varepsilon > 0)(\exists n_0 \in \mathbb{N})(\forall n > n_0)(g(n) < \varepsilon f(n))$
niet kan.
waarbij:
$f(n) = n$
stel:
$g(n) = n^2$
dan geldt voor:
$\varepsilon = 1$
dat:
$(\forall n > n_0)(g(n) < \varepsilon f(n))$
niet geldt.
Dus dan:
$n^2 \notin o(n)$
En kunnen we constateren dat:
$n^2 \in \Omega(n)$

een algoritme met deze complexiteit is bijvoorbeeld bubble sort. Bij deze methode moet bij elke stap twee getallen moeten worden vergeleken totdat het grootste gevonden getal achteraan overblijft. Als alle getallen al geordend zijn hoeven er nooit getallen worden gewisseld. Toch moet elke keer weer een lijst van n-1 worden doorlopen. Dit resulteert in minimaal n stappen. Dus geldt $\Omega(n)$.

c) $\Omega(n^2)$

Er moet gelden ofwel:

$g \notin o(f)$

ofwel:

$(\exists \varepsilon > 0)(\exists^{\infty} n_i)(\frac{g(n_i)}{f(n_i)} > \varepsilon)$

gebruik regel 1, hiervoor moet gelden dat:

$(\forall \varepsilon > 0)(\exists n_0 \in \mathbb{N})(\forall n > n_0)(g(n) < \varepsilon f(n))$

niet kan.

waarbij:

$f(n) \ = n^2$

stel:

$g(n) \ = n^3$

dan geldt voor:

$\varepsilon = 1$

dat:

$(\forall n > n_0)(g(n) < \varepsilon f(n))$

niet geldt

Dus dan:

$n^3 \notin o(n^2)$

En kunnen we constateren dat:

$n^3 \in \Omega(n^2)$

2d matrix van n bij n vermenigvuldigen met een matrix van hetzelfde formaat. Er zijn minimaal n^2 nieuwe elementen nodig om een 2d matrix te vullen , die elk rond de n nieuwe berekeningen vergen. Er geldt dus $\Omega(n^2)$, hier moet echter nog door iemand een werkend algoritme voor worden bedacht.

d) $o(2^n)$

Er moet gelden ofwel:

$(\forall \varepsilon > 0)(\exists n_0 \in \mathbb{N})(\forall n > n_0)(g(n) < \varepsilon f(n))$

ofwel:

$\lim\limits_{n \to \infty} \frac{g(n)}{f(n)} = 0$

Gebruik regel 1, waarbij:
$f(n) = 2^n$
stel:
$g(n) = log(n)$
dan geldt voor:
$\varepsilon = 1$
en:
$n_0 = 0$
dat:
$(\forall n > n_0)(g(n) < \varepsilon f(n))$
geldt.
Dus:
$log(n) \in o(2^n)$

Een voorbeeld van een algoritme van $o(2^n)$ is het toevoegen van een rij nodes in een binary spanning tree. Voor elke set nodes die worden toevoegd kunnen er in de diepte n maximaal 2^{n-1} nodes worden toegevoegd.

e) $\theta(log(n))$
er moet gelden:
$g \in O(f) \cap \Omega(f)$
Begin door te bewijzen dat:
$g \in O(f)$
hiervoor moet gelden:
$(\exists c,\ n_0 \in \mathbb{N})(\forall n > n_0)(g(n) \leq cf(n))$
waarbij
$f(n) = log(n)$
stel:
$g(n) = \sqrt{n}$
dan geldt voor:
$c = 1$
en
$n_0 = 0$
dat
$(\forall n > n_0)(g(n) \leq cf(n))$

dus:

$\sqrt{n} \in O(log(n))$

Bewijs nu dat:
$g \in \Omega(f)$
hiervoor moet gelden ofwel:
$g \notin o(f)$
ofwel:
$(\exists \varepsilon > 0)(\exists^{\infty} n_i)(\frac{g(n_i)}{f(n_i)} > \varepsilon)$
gebruik regel 1, waarbij:
$f(n) = log(n)$
stel:
$g(n) = \sqrt{n}$
dan geldt voor:
$\varepsilon = 1$
dat:
$(\forall n > n_0)(g(n) < \varepsilon f(n))$
niet geldt
Dus dan:
$\sqrt{n} \notin o(log(n))$

En kunnen we constateren dat:
$\sqrt{n} \in \Omega(log(n))$

Hierdoor kunnen we stellen dat:
$g \in O(f) \cap \Omega(f)$
Dus:
$\sqrt{n} \in \theta(log(n))$

Een mogelijk algoritme hiervoor is binary search. Worst-case heeft dit algoritme log(n) operaties nodig. Er zijn nooit meer dan log(n) operaties nodig. Ofterwijl binary search is van orde $O(log(n))$ maar niet van orde $o(log(n))$. Omdat binary search niet van orde $o(log(n))$ is, is het wel van orde $\Omega(log(n))$. En omdat het zowel van orde $\Omega(log(n))$ als orde $O(log(n))$ is, is het ook van orde $\theta(log(n))$

f) $\omega(n)$
Er moet gelden:
$f \in o(g) \Leftrightarrow (\forall \varepsilon > 0)(\exists n_0 \in \mathbb{N})(\forall n > n_0)(f(n) < \varepsilon g(n))$

waarbij:
$f(n) = n$
stel:
$g(n) = n^2$
dan geldt voor:
$\varepsilon = 1$
en:
$n_0 = 0$
dat:
$(\forall n > n_0)(f(n) < \varepsilon g(n))$
geldt.
Dus:
$n^2 \in \omega(n)$

een algoritme met deze complexiteit is bijvoorbeeld selection sort. Deze methode heeft altijd n^2 stappen nodig. Hierboven staat bewezen dat als $g(n) = n^2$ is dat $n^2 \in \omega(n)$

3) Voor $k = 1$:
Neem $n_0 = 1$ en $c = 1$
Dan moet gelden:: $\forall n > 1 \in \mathbb{N}\ (log(n) \leq n)$
$\frac{d}{dn}(log(n)) = \frac{1}{n}$
$\frac{d}{dn}(n) = 1$
Voor $n \geq 1$ geldt $\frac{1}{n} \leq 1$

Dus $log(n) \leq n$ voor $n \geq 1$
Dus $log(n) \in O(n)$
Dus $log^k(n) \in O(n^{1/k})$ voor $k = 1$

Voor een grotere k wordt de waarde van $log^k(n)$ groter terwijl de waarde van $n^{1/k}$ kleiner wordt.

4) Er is gegeven dat

$f \in O(g)$

hieruit volgt

$f \leq c_0 * g$

dan geldt:

$\frac{f}{g} \leq c_0$

nu moet gelden:

$(\exists c,\ n_0 \in \mathbb{N})(\forall n > n_0)(g(n) \leq c_1 f(n))$

waarbij

$f(n)\ = 1$

$g(n) = \frac{f}{g}$

dan geldt voor:

$c_1 = c_0$

en

$n_0 = 0$

dat

$(\forall n > n_0)(g(n) \leq c_1 f(n))$

dus:

$\frac{f}{g} = O(1)$

5)

```
pairs[n];
for(student = 0; student<n; student++){
        skiMin = inf;
        for(ski : h){
                if(l[student] - ski< l[student] - skiMin)
                        skiMin = ski;
        }

        pairs[student] = skiMin;
        h.remove(skiMin);
}
```

Het worst-case scenario voor dit algoritme is 1.5*n operaties. In dat geval moeten alle stappen worden doorlopen. Nadat een student een paar skis toegewezen heeft gekregen worden deze skis uit de verzameling gehaald, hierdoor wordt de binnenste loop elke keer minder lang. Best-case is dit algoritme n operaties kwijt. Dan worden gelijk de juiste skis aan de juiste leerlingen gematcht.