3. feladat - Statisztikai adatok megjelenítése

Feladat

Módosítsd az időjárás widgetet úgy, hogy jelenítse meg a heti hőmérsékletek minimumát, maximumát és átlagát is!

1. Vegyél fel a **contact.html** fájlban három új bekezdést a widgetbe, amelyekben megjelenítheted a hőmérsékleti statisztikákat.
2. A JavaScript fájlban a három számítást három külön függvényben készítsd el, majd a megjelenítésnél ezeket a függvényeket hívd meg!
3. A másik három oldalon is módosítsd a widgetet!

Ismerkedés az algoritmusokkal

## **Algoritmus, avagy al-Hvárizmi kicsit félrefordítva**

Az algoritmus egy feladat, probléma megoldásához vezető lépések, utasítások sorozata. Például egy recept is egy algoritmus, leírja lépésenként, hogyan kell elkészíteni egy ételt. Ha követed pontosan az utasításokat, akkor valószínűleg sikerül neked is az asztalra varázsolni a vacsorát. Ez persze függ attól is, mennyire részletesen tartalmazza a recept az adott étel elkészítéséhez tartozó lépéseket. Egy kezdőnek természetesen minden apró mozzanatot el kell magyarázni, míg egy gyakorlott séfnek elegendő egy nagyobb léptékű leírás. Ebből is látszik, hogy egy problémát nem csak egyféle algoritmussal oldhatsz meg. Nemcsak a lépések mélységében különbözhetnek ezek az algoritmusok, hanem akár teljesen más lépésekből is állhatnak, és a végeredmény mégis ugyanaz.

## **Az algoritmus jellemzői**

Az egyszerű algoritmusokon túl lehetnek akár igen bonyolultak is, mint pl. egy nagy forgalmú reptér irányítása. Az algoritmussal szemben ezért is állítottak néhány követelményt:

* **Véges:** véges számú lépésből áll.
* **Egyértelmű:** a megoldáshoz vezető lépéssorozat tisztán, egyértelműen van megadva, és szigorúan követhető, nem hagy kétségeket.
* **Elvégezhető:** elvárjuk, hogy az algoritmust végre lehessen hajtani, vagyis a lépések elég egyszerűek legyenek, hogy pontosan végre lehessen hajtani.

## **Algoritmusleíró eszközök**

Az algoritmus leírásának módja több tényezőtől függ, például a megoldandó problémától, a célközönségtől. Egy egyszerű algoritmust akár **szövegesen** is leírhatsz pár mondatban röviden, pl. a reggeli kávé elkészítése:

* Bekapcsolom a kávéfőző gépet.
* Ellenőrzőm van-e elég kávé és víz a gépben.
* Amikor felmelegedett, odateszem a csészét és megnyomom az espresso gombot.

Előfordulhat, hogy **képeket** használnak az algoritmus bemutatására vagy önállóan, vagy a szöveges leírás mellé kiegészítésképpen. Gondolj csak a repülőgépeken a Safety On Board leírásokra (ne legyen rá szükséged), például mi a teendő, ha fogy az oxigén az utastérből.

Amíg csak utasításokat kell sorrendben egymás után sorolnom, addig ezek a módszerek teljesen megfelelnek. Viszont amikor már a szekvencián kívül más vezérlő szerkezetet is tartalmaz az algoritmus, akkor szerencsésebb lehet valami olyan leíró eszköz, ami szemléletesebb. A folyamatábra például a folyó szövegnél sokkal kifejezőbb ezekben az esetekben.

## **Vezérlési szerkezetek**

A szekvencián kívül két vezérlési szerkezetet használtunk a folyamatábrán.

* Az egyik az elágazás (szelekció), amikor feltételtől függően más-más irányba halad tovább a vezérlés.
* A másik szerkezet az ismétlés (iteráció), amikor bizonyos esetben visszairányítjuk a vezérlést egy korábbi pontra, mint pl. amikor várakoztunk, amíg fel nem melegedett a gép.
* Az elágazás és ciklus különböző fajtáiról majd a későbbiekben lesz szó.

## **Strukturált algoritmus**

Strukturált az algoritmus, ha csak az alap 3 vezérlőszerkezetet (szekvencia, szelekció, iteráció) használod. Ezeket egymásba is ágyazhatod, de minden ilyen egységnek csak egy kimenete lehet. Vagyis nem húzhatod a nyilakat bárhova.

## **Pszeudo kód, a kis hamis**

Nézzünk egy másik algoritmusleíró módszert, a pszeudo kódot, ami nagyon hasonlít már a programkódra, de nyelvfüggetlen.

KÁVÉFŐZŐ BEKAPCSOLÁSA

IF NINCS ELÉG KÁVÉ THEN

KÁVÉ HOZZÁADÁSA

ENDIF

IF NINCS ELÉG VÍZ THEN

VÍZ HOZZÁADÁSA

ENDIF

WHILE NEM MELEGEDETT FEL

10 MP VÁRAKOZÁS

ENDWHILE

CSÉSZE ODAHELYEZÉSE

GOMB MEGNYOMÁSA

Ebben a pszeudo kódban az angol kifejezéseket használtam, de természetesen megengedett a magyar nyelvű változat is.

Nagyon fontos az indentálás (beljebb kezdés), hogy lásd az algoritmus szerkezetét, ezáltal látod az alapstruktúrák egymásba ágyazását.

A pszeudo kódban könnyebb betartani, hogy strukturált algoritmust írj, a folyamatábrán könnyű a nyilakat úgy húzni, hogy elveszítsd a strukturáltságot. Napjainkban is van azért helye, például az UML-ben (Unified Modeling Language) is találunk a folyamatábrához hasonlót, csak ott aktivitás diagramnak nevezik. Az UML tulajdonképpen egy vizuális modellező nyelv, ami nagyon hasznosnak bizonyul a szoftverek készítésének folyamatában, leginkább a kezdeti analizáló és tervező fázisban. Az objektum orientált programozás tanulásakor használunk majd osztálydiagramot is, ami szintén az UML része.

Természetesen az itt bemutatottakon kívül még más algoritmusleíró eszközök is léteznek, pl. a struktogram, de ezeket most hagyjuk pihenni.

## **Használandó vezérlési szerkezetek a programkódban**

Mivel minden algoritmus felépíthető három vezérlési szerkezet használatával, a programkódban is csak ezeket kell most használnod:

* szekvencia,
* szelekció,
* iteráció.

Van még az ugrás, de arra most nincs szükség.

# Elemi algoritmusok

## **Elemi algoritmusok**

Van néhány olyan folyamat a programozás során, amire az idők során kialakultak a hatékony technikák. Most ezeket mutatom meg.

**Ismétlés:** Az algoritmus azon lépések sorozata, amelyek egy probléma megoldásához vezetnek.

## **Az algoritmus**

Tehát mindig úgy kezdődik, hogy van egy problémád. Ezt a problémát általában sokféleképpen meg lehet oldani, így ahány féle megoldás létezik, annyi algoritmus van hozzá. Jöjjön egy példa!

**Algoritmus bankkártyás fizetésre:** milyen lépésekből áll amikor kártyával fizetsz a boltban?

1. Előveszed a bankkártyát.
2. Ellenőrzöd az összeget a kijelzőn.
3. Hozzáérinted a kártyát a terminálhoz.
4. Megvárod a hangjelzést.
5. Elteszed a kártyát.

Ez az öt lépés gyakorlatilag egy algoritmus. A minimálisan szükséges lépéseket tartalmazza a megoldáshoz. Most olyan mélységben nem megyünk bele a dologba, hogy közben lélegezni is kell, mert arra nyilván van egy létfenntartást szolgáló másik algoritmus. :) Ki lehetne terjeszteni a teljes vásárlás folyamatára is, de abban az esetben olyan lépések is lennének benne, mint mondjuk betesszük a terméket a kosárba, vagy távozunk a boltól.

## **Nevezetes algoritmusok**

Azért hívják őket nevezetesnek, mert gyakran kell őket használni. Olyan programozási folyamatokat írnak le, amelyek nagyon gyakoriak a hétköznapi munka során. Most csak röviden áttekintjük őket, de részletesen is foglalkozunk majd velük, illetve begyakoroljuk őket.

## **Pszeudokód**

Mielőtt azonban tanulmányoznánk az algoritmusokat, ismerkedjünk meg a pszeudokód fogalmával! Itt az egyes lépéseket egyszerű, beszélt nyelven írjuk le. Azért jó, mert csak az algoritmust adjuk meg, és utána gyakorlatilag bármilyen programozási nyelven el tudjuk készíteni a végleges kódot. Azaz nyelvfüggetlen. Például így néz ki egy ciklus, amellyel sokat fogunk még dolgozni:

CIKLUS AMÍG van még szám, ADDIG

szám = következő elem

KI: szám

CIKLUS VÉGE

Ha elemzed a fenti kódot, azt látod, hogy addig fog a ciklus futni, amíg nem ér végig a számokon, amik mondjuk egy tömbben is lehetnek. Minden ciklusmagban beállítja a számot és kiírja az értékét, azaz lényegében kiírja a tömb vagy lista elemeit.

## **Összegzés tétele**

A feladat egyszerű, egy sorozat elemeit kell összesíteni. A pszeudokód:

összeg = 0

CIKLUS AMÍG van még szám, ADDIG

szám = következő elem

összeg = összeg + szám

CIKLUS VÉGE

## **Számlálás tétele**

Itt össze kell számolni, hogy egy bizonyos feltétel hány elemre igaz.

db = 0

CIKLUS AMÍG van még szám, ADDIG

szám = következő elem

HA igaz a feltétel szám-ra, AKKOR

db = db + 1

FELTÉTEL VÉGE

CIKLUS VÉGE

## **Szélsőérték keresése (maximum vagy minimum)**

A dolog egyszerű. Van mondjuk egy számsorozatod, és meg kell mondanod, hogy melyik a legkisebb vagy legnagyobb szám, azaz a szélsőértékeket keresed.

legnagyobb = első elem

CIKLUS AMÍG van még szám, ADDIG

szám = következő elem

HA szám < legnagyobb, AKKOR

legnagyobb = szám

FELTÉTEL VÉGE

CIKLUS VÉGE

## **Eldöntés tétele**

Megvizsgálod, szerepel-e olyan elem a sorozatban, melyre igaz a feltétel. Például: prímszám-e. Ahogy találsz egy osztót, ami nem önmaga és nem 1, tudod hogy nem az.

találat = HAMIS

CIKLUS AMÍG van elem ÉS NEM találat

szám = következő elem

HA igaz a feltétel szám-ra, AKKOR

találat = IGAZ

FELTÉTEL VÉGE

CIKLUS VÉGE

## **Egyéb algoritmusok**

A későbbiekben más algoritmusokkal is foglalkozunk majd, de a fenti négy képezi a programozási elmélet egyik alapkövét, ezért ezeket addig nyomjuk amíg a véreddé és a húsoddá fognak válni :)))

# Összegző és számláló algoritmusok

## **Összegző és számláló algoritmusok**

Itt a nevezetes algoritmusok vagy programozási tételek közül az első kettő kerül terítékre. Nem lesz bonyolult, de kicsit hivatalosabban fogalmazok majd, ne ijedj meg.

## **Összegzés**

Először az elméletet. Így nézett ki a pszeudokód:

összeg = 0

CIKLUS AMÍG van még szám, ADDIG

szám = következő elem

összeg = összeg + szám

CIKLUS VÉGE

### **Alapvető megállapítások**

* Az "akkumulátor" változó az, amelyikben összegyűlik, akkumulálódik az eredmény, ezt most összeg néven tüntettük fel. Ezt először nulláznod kell, utána minden feldolgozott számot hozzá kell adnod. Minden iteráció végén az addig látott számok összegét fogja így tartalmazni. Ha esetleg egyszer sem hajtódott volna végre a ciklus, akkor pedig nullát.
* A ciklus ebben az esetben a teljes sorozaton végig kell hogy iteráljon, mivel az összes számot össze szeretnéd adni.
* A pszeudokódban külön kiemelheted a következő számot egy változóba szám néven, majd ezt adod hozzá az összeghez. Ez a konkrét implementációnál, azaz amikor valamilyen nyelven lekódolod a példát, már egy sorban is könnyen megoldható.

### **JavaScript kód**

var numericArray = [1, 3, 2, 6, 5, 3, 4, 6, 7];

var amount = 0;

for (var i = 0; i < numericArray.length; i++) {

amount += numericArray[i];

}

console.log(amount);

### **A kód magyarázata**

* Az első és utolsó sor, azaz a tömb létrehozása és a console-ra való kiírás nem volt benne a pszeudokódban, ezek a konkrét implementáció során már szükségesek.
* Az algoritmus megvalóstása négy sor csupán, a konkrét implementáció általában rövidebb a pszeudokódnál.
* Az akkumulálásra, azaz az összeg összegyűjtésére az "amount" nevű változót definiáltam, hoztam létre.
* A ciklus egy egyszerű for ciklus. Azért választottam ezt, mert szinte az összes nyelvben ez a szintaxisa. Az iterációk száma megegyezik a tömb elemeinek a számával.
* A ciklusmagban a "+=" rövidítést használtam, ez az összetett operátor mindig a jobb oldalon álló értékkel növeli a változó értékét.

**Fontos:** Az akkumulátor változót mindig a cikluson kívül hozzuk létre és inicializáljuk!

## **Számlálás**

Itt is jöjjön az elmélet:

db = 0

CIKLUS AMÍG van még szám, ADDIG

szám = következő elem

HA igaz a feltétel szám-ra, AKKOR

db = db + 1

FELTÉTEL VÉGE

CIKLUS VÉGE

### **Alapvető megállapítások**

* Nagyon hasonló az összegzéshez, két dologban különbözik tőle.
* A db nevű változó nem az összeget tartalmazza, hanem az elemek számát.
* Csak azokat az elemeket számolja bele az eredménybe, amelyekre egy adott feltétel igaz.

### **JavaScript kód**

var numericArray = [1, 3, 2, 6, 5, 3, 4, 6, 7];

var count = 0;

for (var i = 0; i < numericArray.length; i++) {

if (numericArray[i] > 3) {

count++;

}

}

console.log(count);

### **A kód magyarázata**

* A megfelelő elemek számát a "count" nevű változóba gyűjtöttem.
* Az összes elemen végig iterál a ciklus.
* Ha az adott elem értéke nagyobb, mint 3, akkor növeli a "count" változó értékét eggyel.

# Szélsőértékek keresése

## **Szélsőérték és eldöntés algoritmusa**

Most pedig megmutatom a szélsőérték keresés és az eldöntés algoritmusát.

## **Szélsőérték**

Ha van egy nem üres sorozat, akkor két szélső érték lesz benne, a legkisebb és a legnagyobb. Szokták maximum vagy minimum kiválasztásnak is hívni ezt az eljárást. Előfeltétele, hogy az elemek egymással összehasonlíthatóak legyenek, azaz el tudjuk dönteni, hogy két elem közül melyik a nagyobb. Először a pszeudokód:

legnagyobb = első elem

CIKLUS AMÍG van még elem, ADDIG

elem = következő elem

HA elem > legnagyobb, AKKOR

legnagyobb = elem

FELTÉTEL VÉGE

CIKLUS VÉGE

### **Alapvető megállapítások**

* A szélsőérték keresés során mindig azt feltételezzük, hogy az első elem a kiválasztott, azaz a legkisebb vagy legnagyobb, attól függően, hogy melyiket keressük. Ezt jelenti, hogy a legnagyobb az első elem.
* Ha nem rendezett az elemek listája, akkor minden elemen végig kell iterálnunk egy ciklussal, mivel nem tudjuk, hogy vannak-e még nagyobb vagy kisebb elemek a listában.
* Az eldöntés tételéhez nagyon hasonlóan, itt is van egy feltétel a ciklusmagban. Ez a feltétel azt vizsgálja, hogy a legutóbb maximumnak tartott értékhez képest a jelenlegi érték nagyobb-e. Ha igen, akkor a jelenlegi értéket állítja be legnagyobb értéknek.
* A minimum kiválasztás gyakorlatilag megegyezik ezzel, csupán a relációs jelet kell megfordítanunk a ciklusmagban, amikor vizsgáljuk a feltételt.

### **JavaScript kód**

let numericArray = [1, 3, 2, 6, 5, 3, 4, 6, 7];

let biggest = numericArray[0];

for (let i = 0; i < numericArray.length; i++) {

if (numericArray[i] > biggest) {

biggest = numericArray[i];

}

}

console.log(biggest);

### **A kód magyarázata**

* Feltételezzük, hogy az első elem a legnagyobb, ezért be is állítjuk a tömb első elemét a "biggest" értékeként.
* A ciklussal a teljes tömbön végig iterálunk.
* Ha az adott elem nagyobb, mint a feltételezett legnagyobb, azaz a "biggest", akkor a "biggest"-et felülírjuk az elemre.
* Végül logoljuk az eredményt.

## **Eldöntés**

Itt is kezdjük az elmélettel! Az alapfeladat kétféle lehet. Vagy azt akarjuk eldönteni, hogy egy sorozatban van-e valamilyen tulajdonsággal rendelkező elem, vagy azt, hogy minden elem rendelkezik-e az adott tulajdonsággal. Mindkét esetben logikai érték lesz a válasz. Lássuk először az első esetet:

találat = HAMIS

CIKLUS AMÍG van elem ÉS NEM találat

elem = következő elem

HA keresett tulajdonságú az elem, AKKOR

találat = IGAZ

FELTÉTEL VÉGE

CIKLUS VÉGE

### **Alapvető megállapítások**

* A "találat" változót hamis értékkel definiáljuk, azaz azt felételezzük, hogy nincs találat.
* A ciklus feltételében nem csak az elemek szerepelnek, azt is vizsgáljuk, hogy van-e találat. Így ha már van találat, akkor a ciklus nem fog többet lefutni.
* Ha az aktuális elem a keresett tulajdonságú, akkor a "találat" igazzá válik, és a ciklus futása leáll.
* Ha egyetlen elemre sem igaz a feltétel, akkor a ciklus az összes elemet bejárja, és utána a "találat" változó hamis marad.

### **JavaScript kód**

Lássunk egy példát: arra keressünk a választ, hogy a számok sorozatában van-e hatos.

let numericArray = [1, 3, 2, 6, 5, 3, 4, 6, 7];

let find = 6;

let found = false;

for (let i = 0; i < numericArray.length && !found; i++) {

if (numericArray[i] == find) {

found = true;

}

}

console.log(found);

### **A kód magyarázata**

* Az "i" változó fogja tárolni, hogy éppen hol járunk a tömb iterálása során.
* A "find" tartalmazza a keresett értéket.
* A "found" változó tárolja, hogy megtaláltuk-e a keresett értéket.
* Ha a vizsgálat igaz, azaz a tömb aktuálisan vizsgált elemének értéke megegyezik a keresett értékkel, akkor a "found" változót true értékre állítjuk. Ennek hatására a ciklus nem fog többet lefutni, mivel a feltétele hamissá válik.
* Végül logoljuk az eredményt.

Most nézzük meg azt az esetet, amikor az a kérdés, hogy minden elem rendelkezik-e az adott tulajdonsággal. Ha az éppen vizsgált elem megfelelő, az még nem ad nekünk választ, tovább kell vizsgálódnunk. Ha az aktuális elem nem megfelelő, akkor viszont biztosak lehetünk benne, hogy a válasz hamis, azaz nem minden elem rendelkezik az adott tulajdonsággal. Ez alapján a módosított algoritmus a következő:

mind = IGAZ

CIKLUS AMÍG van elem ÉS mind

elem = következő elem

HA nem a keresett tulajdonságú az elem, AKKOR

mind = HAMIS

FELTÉTEL VÉGE

CIKLUS VÉGE

### **Alapvető megállapítások**

* A "mind" változót igaz értékkel definiáljuk, azaz azt felételezzük hogy minden elem megfelelő.
* A ciklus feltételében azt is vizsgáljuk, hogy még mindig igaz-e mindre. Így ha már valamelyik elem nem volt megfelelő, akkor a ciklus nem fog többet lefutni.
* Ha az aktuális elem nem az adott tulajdonságú, akkor a "mind" hamissá válik, és a ciklus futása leáll.
* Ha minden elemre igaz a feltétel, akkor a ciklus az összes elemet bejárja, és utána a "mind" változó igaz marad.

### **JavaScript kód**

Lássunk egy példát: arra keressünk a választ, hogy számok sorozatában mind pozitív-e.

let numericArray = [1, 3, -2, 6, 5, 3, -4, 6, 7];

let all = true;

for (let i = 0; i < numericArray.length && all; i++) {

if (numericArray[i] <= 0) {

all = false;

}

}

console.log(all);

### **A kód magyarázata**

* Az "i" változó fogja tárolni, hogy éppen hol járunk a tömb iterálása során.
* Az "all" változó tárolja, hogy mind pozitív-e.
* Ha a vizsgálat igaz, azaz a tömb aktuálisan vizsgált elemének értéke nem pozitív, akkor az "all" változót false értékre állítjuk. Ennek hatására a ciklus nem fog többet lefutni, mivel a feltétele hamissá válik.
* Végül logoljuk az eredményt.

# Debugging

## **Debugging**

Sok fejfájást okoz hogy megtaláld a hibát a kódodban. Szerencsére erre vannak fejlett eszközök, ezeket debuggereknek vagy hibakeresőknek szokták nevezni. Én most a Google Chrome beépített debuggerét mutatom meg neked.

## **Chrome debugger**

Mivel JavaScript nyelven dolgozunk és böngészőben, adja magát hogy a böngésző beépített debuggerét használjuk.

* F12 gombbal nyisd meg a fejlesztői eszköztárat.
* A **Sources** fülre navigálj az eszköztáron belül. Itt szépen csoportosítva látod az összes fájlt, képet, JavaScript-et amit az oldal használ:
* Válassz ki egy JavaScript fájlt és ekkor megjelenik a jobb oldali ablakban a fájl tartalma. Ez gyakorlatilag az a script amit a böngésző futtat. Származhat harmadik féltől is, vagy lehet a magad által készített kód:
* Ha a sorok számára kattintasz a bal oldalon, akkor megjelenik egy kék nyíl, és ezzel létrehozol egy úgynevezett töréspontot (breakpoint), ahol a kód futtatása során meg fog állni a böngésző és addig nem megy tovább, amíg nem avatkozol be:
* Ezután egyszerűen frissítsd le az oldalt. Ha a végrehajtás rákerül arra a sorra amit kijelöltél, akkor a böngésző megállítja a kód futtatását és meg tudod vizsgálni a változók tartalmát. Ha nem kerül a végrehajtás arra a sorra, annak az lehet az oka, hogy valamilyen esemény hatására fut le a kódrészlet, vagy esetleg olyan feltételhez van kötve a futtatás ami nem teljesül:
* Látszik a képen, hogy a currentLessons egy NodeList 106 elemmel, és az i változó pillanatnyi értéke épp 1. Mivel rávittem az egeret a this.value változóra, ami egy input mező értéke, így azt is szépen megmutatja a Chrome egy szövegbuborékban.
* Miután befejezted a kód vizsgálatát, a bal alsó sarokban vagy a képernyő közepén megjelenő kék léptető gombbal tudsz továbblépni a következő töréspontra vagy ennek hiányában befejezni a debuggolást:

## **Hiba esetén mit tegyünk?**

1. Ne essünk kétségbe. Csak az nem hibázik aki nem dolgozik. Sokkal jobb, ha mi vesszük észre a hibát mintha a megrendelő.
2. Nyissuk meg a fejlesztői eszköztárat és nézzük meg a konzolt. Ha ott piros hibaüzenetet látunk, akkor általában egy hivatkozás is van, hogy melyik fájl hányadik sorában keletkezett a hiba. Itt már elhelyezhetünk ízlés szerint töréspontokat a megfelelő sorokba, amennyiben szükséges.
3. Erre a hivatkozásra kattintva egyből a Sources panel adott sorára ugrik az eszköztár. Itt megnézhetjük a kódot és ellenőrizhetjük hogy mi váltotta ki a hibát.
4. Ne felejtsük bent a töréspontokat miután a hibát kijavítottuk és ellenőriztük a helyes futást.