



# CSS Grid

- do tworzenia layoutów (interfejsów użytkownika)
- pozycjonowanie elementu (elementów) w innym elemencie

Czyli jak Flexbox, ale trochę inaczej.

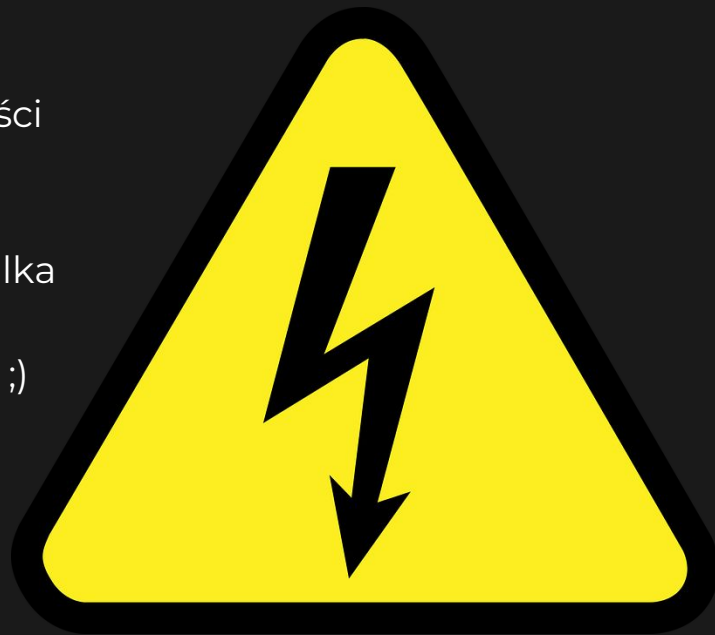


# Uwaga wstępna

Są takie sytuacje w CSS Grid, że mam wątpliwości czy sami twórcy na pewno go ogarniają ;)

Podejrzewam, że spotkało się kilka koncepcji, kilka trudności trzeba było rozwiązać, może podjąć trudne decyzje - tylko czy na pewno na trzeźwo ;)

Ale tak poważnie to w 98% jest super.



# Co to jest layout?

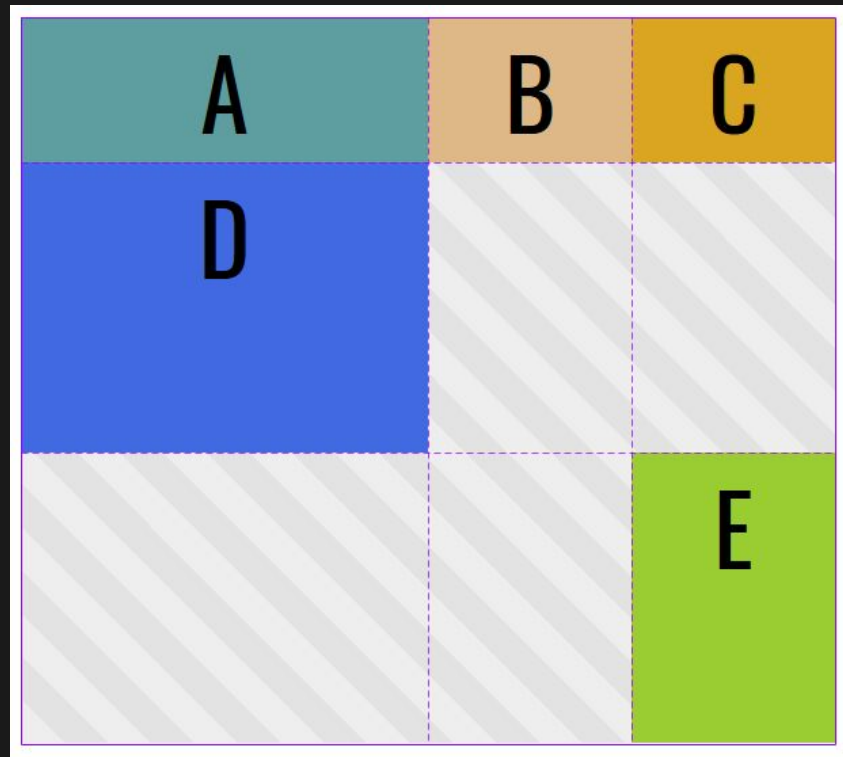
- interfejs strony internetowej
- struktura, szablon, układ strony

Możemy mówić o layoucie strony/aplikacji oraz o layoucie komponentu, czyli rozmieszczeniu elementów wewnątrz komponentów.



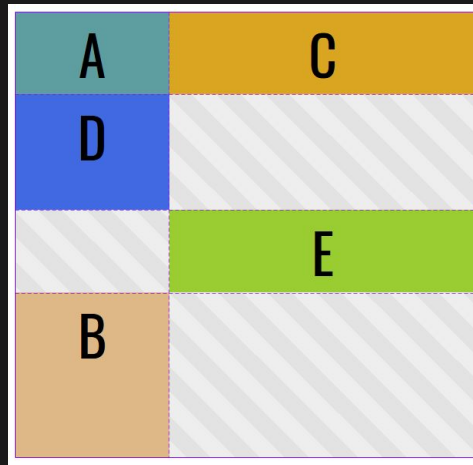
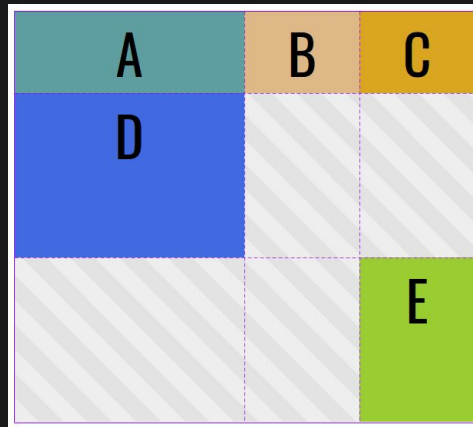
# O co chodzi z CSS Grid

Idea, która stoi za CSS Grid, to oprzeć layout o strukturę **siatki**. Siatka składa się z kolumn i wierszy (jest **dwuwymiarowa**). Najmniejszą funkcjonalną częścią siatki jest **komórka**.



# CSS GRID

- Pozwala deweloperowi precyzyjnie określić strukturę siatki i powierzchnie dla elementów lub wytycza je automatycznie zgodnie z przyjętymi regułami.
- Posiada właściwości ułatwiające pracę z układami responsywnymi oraz umożliwia przetworzenia struktury siatki przy różnych rozdzielczościach (świetne z Media Queries)
- Daje wiele “dobrych dróg” do celu. Co prawda czyni to CSS Grid bardziej skomplikowanym. Na szczęście, nie trzeba poznawać wszystkich jego możliwości by sprawnie z niego korzystać.





# CSS Grid a Flexbox



# CSS Grid i Flexbox

CSS Grid - kontrola layoutu za pomocą **dwuwymiarowej siatki**.

Flexbox - kontrola layoutu za pomocą **jednowymiarowej osi**.

Świetnie się uzupełniają. Na stronie możesz umieszczać oba rozwiązania. Co więcej możesz zagnieżdżać je w sobie.


Nazwy modułów: CSS Grid Layout i CSS Flex Box Layout



# CSS Grid - 90% i więcej (sierpień 2019)

## CSS Grid Layout (level 1) - CR

Method of using a grid concept to lay out content, providing a mechanism for authors to divide available space for layout into columns and rows using a set of predictable sizing behaviors. Includes support for all `grid-*` properties and the `fr` unit.

Usage % of all users  ?  
Global 91.11% + 2.15% = 93.26%  
unprefixed: 91.11%

| Current aligned | Usage relative | Date relative | Apply filters | Show all  | ?       |              |              |                   |                    |                |                    |                     |           |                        |            |         |  |  |  |  |
|-----------------|----------------|---------------|---------------|-----------|---------|--------------|--------------|-------------------|--------------------|----------------|--------------------|---------------------|-----------|------------------------|------------|---------|--|--|--|--|
| IE              | Edge *         | Firefox       | Chrome        | Safari    | Opera   | iOS Safari * | Opera Mini * | Android Browser * | Blackberry Browser | Opera Mobile * | Chrome for Android | Firefox for Android | IE Mobile | UC Browser for Android | Sams Inter |         |  |  |  |  |
|                 |                | 2 - 39        | 4 - 28        |           |         |              |              |                   |                    |                |                    |                     |           |                        |            |         |  |  |  |  |
|                 |                | 40 - 51       | 29 - 56       |           | 10 - 27 |              |              |                   |                    |                |                    |                     |           |                        |            |         |  |  |  |  |
| 6 - 9           | 12 - 15        | 52 - 53       | 57            | 3.1 - 10  | 28 - 43 | 3.2 - 10.2   |              |                   |                    |                |                    |                     |           |                        |            | 4 - 5   |  |  |  |  |
| 10              | 16 - 17        | 54 - 67       | 58 - 75       | 10.1 - 12 | 44 - 60 | 10.3 - 12.1  |              | 2.1 - 4.4.4       | 7                  | 12 - 12.1      |                    |                     | 10        |                        |            | 6.2 - 8 |  |  |  |  |
| 11              | 18             | 68            | 76            | 12.1      | 62      | 12.3         | all          | 67                | 10                 | 46             | 75                 | 67                  | 11        | 12.12                  | 9.2        |         |  |  |  |  |
|                 | 76             | 69 - 70       | 77 - 79       | 13 - TP   |         | 13           |              |                   |                    |                |                    |                     |           |                        |            |         |  |  |  |  |

Jak sobie z tym radzić opowiem w sekcji poświęconej dostępności i optymalizacji





# Flexbox > 95% (sierpień 2019)

## CSS Flexible Box Layout Module - CR

Method of positioning elements in horizontal or vertical stacks.  
Support includes all properties prefixed with `flex`, as well as `display: flex`, `display: inline-flex`, `align-content`, `align-items`, `align-self`, `justify-content` and `order`.

| Usage       | % of all users          | ? |
|-------------|-------------------------|---|
| Global      | 95.72% + 2.96% = 98.68% |   |
| unprefixed: | 95.56% + 1.94% = 97.5%  |   |

| Current aligned | Usage relative | Date relative | Apply filters | Show all | ?       |              |              |                   |                    |                |                    |                     |           |                        |             |
|-----------------|----------------|---------------|---------------|----------|---------|--------------|--------------|-------------------|--------------------|----------------|--------------------|---------------------|-----------|------------------------|-------------|
| IE              | Edge *         | Firefox       | Chrome        | Safari   | Opera   | iOS Safari * | Opera Mini * | Android Browser * | Blackberry Browser | Opera Mobile * | Chrome for Android | Firefox for Android | IE Mobile | UC Browser for Android | Sams Intern |
|                 |                | 1 2-21        | 1 4-20        | 1 3.1-6  | 10-11.5 | 1 3.2-6.1    |              |                   |                    |                |                    |                     |           |                        |             |
| 6-9             |                | 2 22-27       | 21-28         | 6.1-8    | 15-16   | 7-8.4        |              | 1 2.1-4.3         |                    | 12             |                    |                     |           |                        |             |
| 2 4 10          | 12-17          | 28-67         | 29-75         | 9-12     | 17-60   | 9-12.1       |              | 4.4-4.4.4         | 1 7                | 12.1           |                    |                     | 2 10      |                        | 4-8         |
| 4 11            | 18             | 68            | 76            | 12.1     | 62      | 12.3         | all          | 67                | 10                 | 46             | 75                 | 67                  | 11        | 12.12                  | 9.2         |
|                 | 76             | 69-70         | 77-79         | 13-TP    |         | 13           |              |                   |                    |                |                    |                     |           |                        |             |

# CSS Grid i Flexbox

Flexbox - to dystrybucja wolnego miejsca w kontenerze.

CSS Grid - to umiejscawianie elementów na powierzchni siatki.

CSS Grid wymaga od nas więcej pracy. Z tego powodu, a także ze względu na wsparcie w przeglądarkach (Grid bardzo dobre, Flexbox doskonałe), polecam najpierw przemyśleć czy wystarczy nam jednowymiarowy Flexbox. Jeśli nie, to wybierzmy CSS Grid.

W wielu przypadkach ten sam efekt osiągniemy za pomocą Flexboxa i CSS Grida.





# Najważniejsze kategorie związane z CSS Grid



# Definicje

- kontener siatki
- element siatki
- linie siatki
- ścieżki (tory)
- komórka siatki
- obszar (powierzchnia) siatki
- przerwy w siatce



# Kontener (Grid Container)

```
<div class="grid-container">
  <div class="element1">A</div>
  <div class="element2">B</div>
</div>

.grid-container {
  display: grid; /* kontener siatki */
}
```

Element HTML tworzący siatkę.

Tworzy granicę dla siatki.

Do powstania siatki sama deklaracja kontenera to za mało. Potrzebne są także elementy siatki lub jawne zadeklarowanie struktury siatki.

Na jednej stronie możesz umieścić wiele kontenerów siatki.



# Elementy siatki (Grid Item)

```
<div class="grid">
  <div class="element1">
    <div class="text">
      tekst
    </div>
  </div>
  <div class="element2">B</div>
</div>
```

```
.grid {
  display: grid;
}
```

Element(y) siatki, to element(y) HTML będący “dzieckiem” kontenera. Elementem siatki jest domyślnie każdy element bezpośrednio zagnieżdżony w kontenerze.

Kolejni potomkowie (bardziej zagnieżdżone elementy w strukturze HTML) zachowują się już “normalnie” tzn. siatka nie ma na nie wpływu. Działa to tak samo jak w znanym Ci już Flexboxie (kontener i elementy elastyczne).



# Linie (Grid Line)

Linie to wirtualne byty (nie mają wielkości i nie oddziałują z elementami siatki) który określają początek i koniec kolumn i wierszy. Gdy mamy jedną kolumnę istnieją 4 linie: dwie linie pionowe (na początku kolumny i na końcu) i dwie linie poziome.

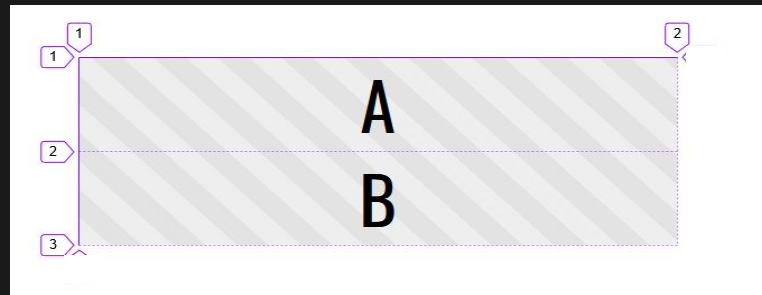
```
<div class="grid">  
  <div class="e1">Element</div>  
</div>
```



# Linie (Grid Line) - przykład z dwoma wierszami

Gdy mamy dwa wiersze, to tworzą się już trzy linie poziome i dwie linie pionowe. Dzięki nim powstają “koordynaty” wierszy i kolumn. Każda kolumna/wiersz ma linię początkową i końcową. Linia końcowa jednego wiersza/kolumny może być oczywiście linią początkową kolejnego wiersza/kolumny.

```
<div class="grid">  
  <div class="e1">A</div>  
  <div class="e2">B</div>  
</div>
```

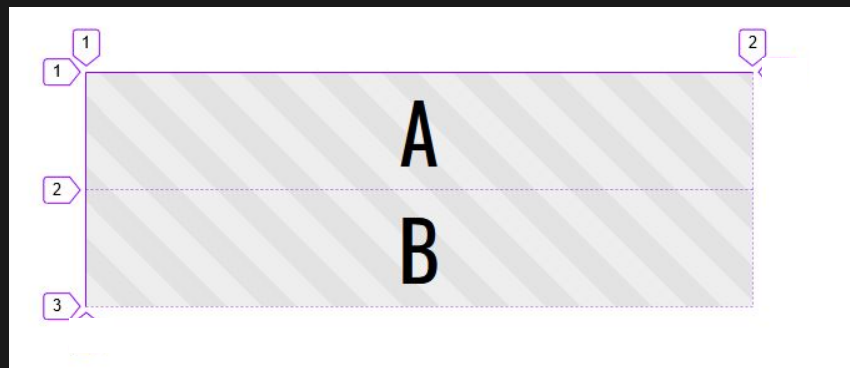




# Linie (Grid Line)

Na tym etapie pamiętaj, że linie “rysują” układ kolumn i wierszy oraz mają swoją nazwę domyślną w postaci liczby. Linie są też podstawowym sposobem w jaki definiowane są (jawnie lub automatycznie) pozycji elementu siatki w siatce.

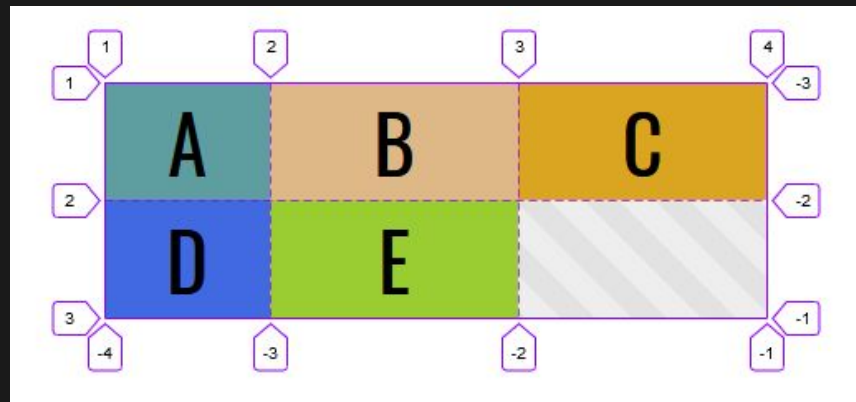
```
<div class="grid">  
  <div class="e1">A</div>  
  <div class="e2">B</div>  
</div>
```



# Linie (Grid Line)

Linie liczymy (nazywamy) od liczby 1. Od lewej strony dla kolumn i od góry dla wierszy. Pamiętaj jednak, że równocześnie linie liczone są od przeciwnej strony, tyle tylko, że z minusami (od -1 od prawej strony dla kolumn i od dołu dla wierszy).

Linia zawsze znajduje się przed pierwszym wierszem/kolumną i za ostatnim wierszem/kolumną. Tak więc linii poziomych jest zawsze o jeden więcej niż wierszy, a linii pionowych o jeden więcej niż kolumn.



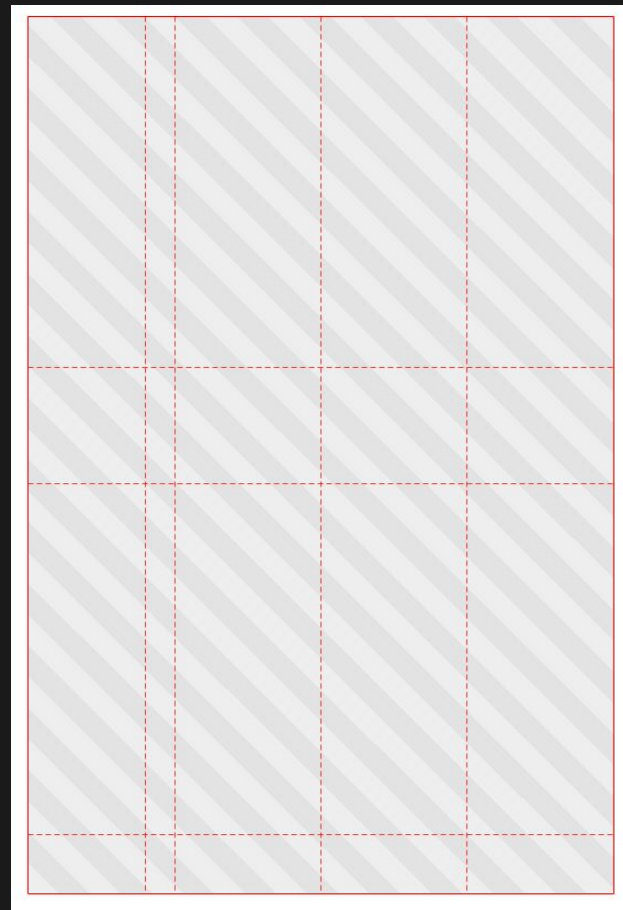
# Ścieżki (Grid Tracks)

## - wiersze i kolumny

Siatka w CSS Grid oparta jest na wierszach i kolumnach (określamy je ścieżkami czy torami). By mówić o siatce musi wystąpić co najmniej jedna kolumnę i jeden wiersz. Wiersze i kolumny są “wytyczane” przez linie.

Dana kolumna czy dany wiersz zawsze ma taką samą wielkość na całym swoim torze (ścieżce). Tor swoją długością zawsze obejmuje cały kontener (w przypadku wiersza będzie to szerokość, a w przypadku kolumny wysokość)

*W przykładzie obok mamy 5 kolumn (columns) i 4 wiersze (rows).*



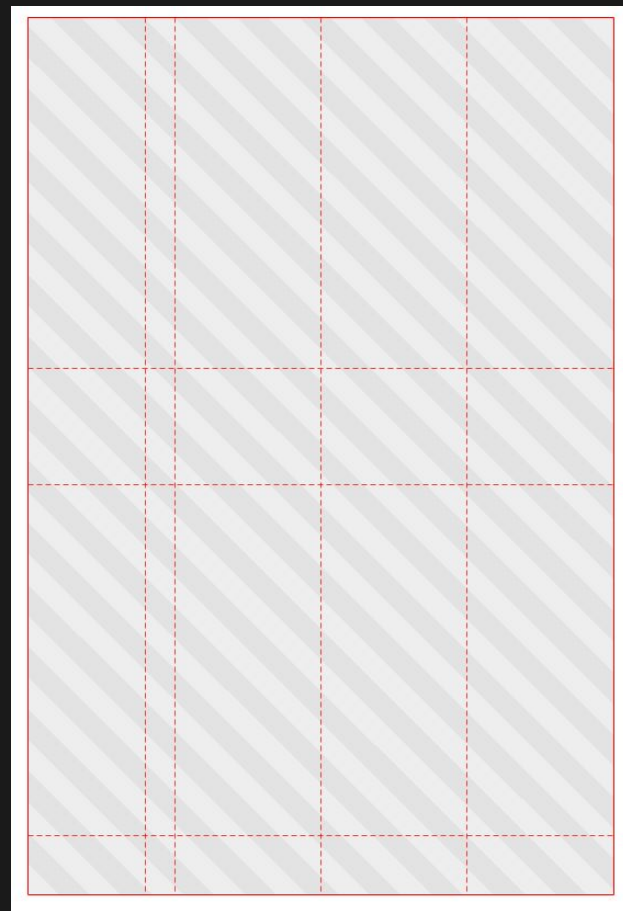
# Ścieżki (Grid Tracks)

## - wiersze i kolumny

Kiedy mówimy o wielkości danego wiersza/kolumny to mówimy:

- w przypadku wiersza o jego wysokości
- w przypadku kolumny o jej szerokości

Te rozmiary mogą być różne dla poszczególnych wierszy i kolumny. Natomiast szerokość każdego wiersza będzie zawsze w danej siatce taka sama, podobnie jak wysokość każdej kolumny.



# Komórka (Grid Cell)

Komórka jest najmniejszym funkcjonalnym elementem siatki - powstaje na przecięciu kolumny i wiersza.

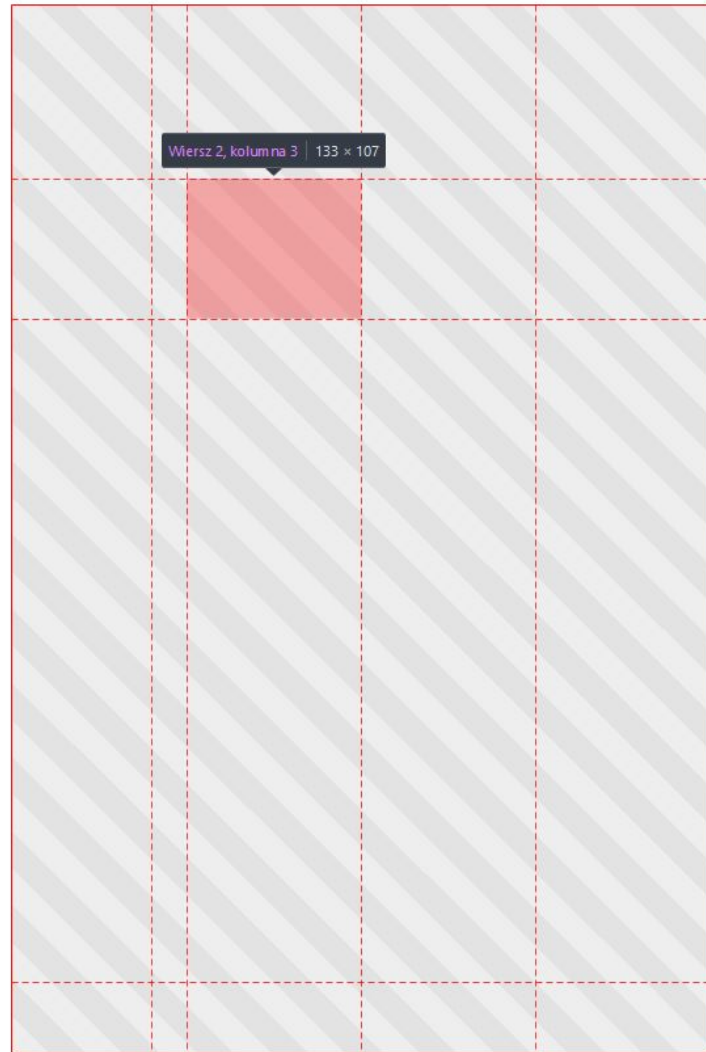
Komórka siatki, to ta część siatki na której może być umieszczony element siatki (czyli bezpośredni potomek kontenera - jeśli patrzymy od strony struktury HTML).



# Komórka (Grid Cell)

Komórka powstaje na przecięciu kolumny i wiersza. Jest wydzielana z każdej strony przez linię - dlatego można ją precyzyjnie zaadresować poprzez numer linii (pionowych w przypadku kolumn i poziomych w przypadku wierszy). Komórki można też nazwać, co jest bardzo popularną formą pracy z CSS Grid.

W jednej komórce może się znaleźć więcej niż jeden element siatki. z drugiej strony nie każda komórka musi być zajmowana przez element siatki.

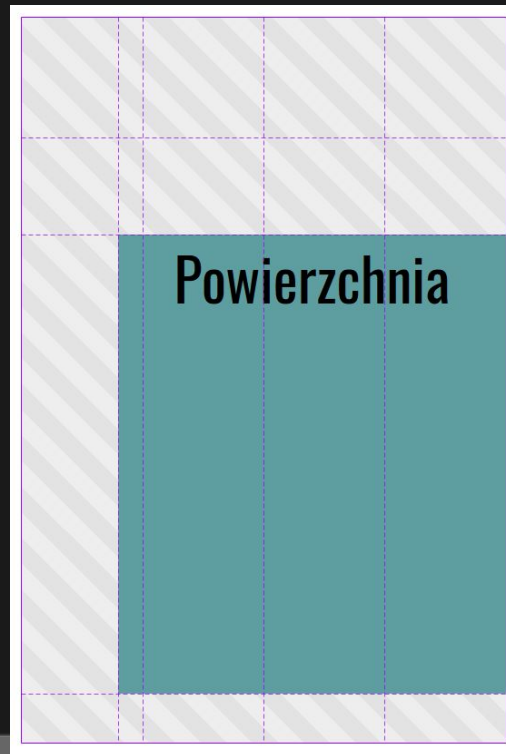


# Powierzchnia (obszar) siatki - Grid Area

Powierzchnia (obszar) siatki - zbiór komórek (minimalnie jedna komórka) na których umieszcza się element siatki.

Powierzchnia siatki to przestrzeń na umieszczenie elementu, Ma jedną lub więcej komórek.

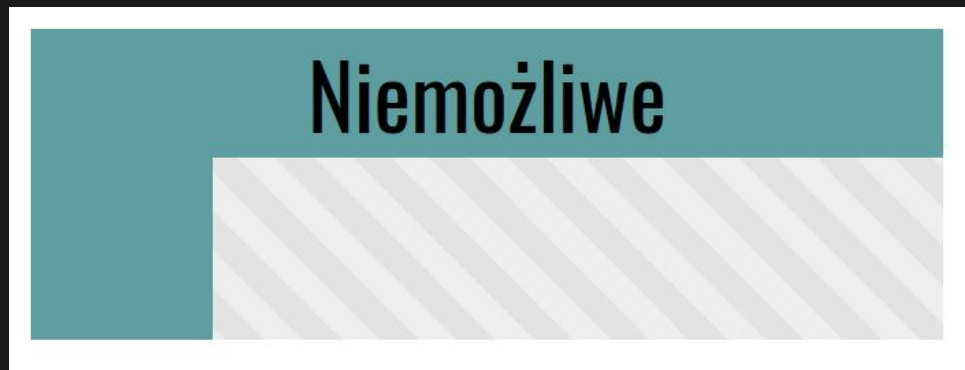
Największą powierzchnię siatki stanowi cała siatka (obejmuje wszystkie komórki + ewentualne przerwy), a najmniejszą jedna komórka.



# Powierzchnia siatki

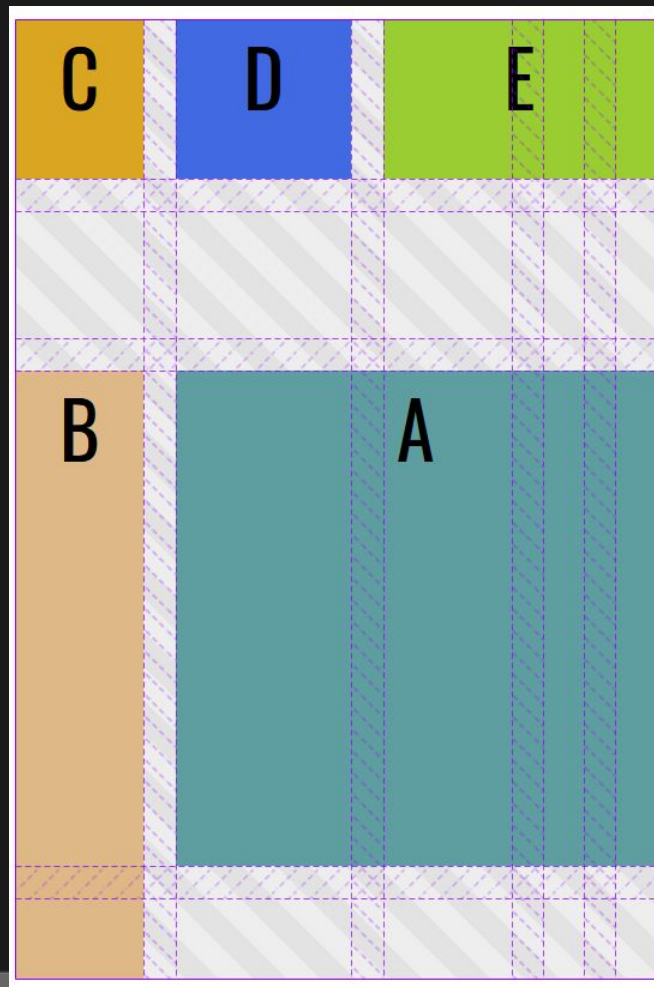
Powierzchni siatki ma zawsze charakter prostokąta.

Gdy wytyczamy pozycję elementu siatki w siatce lub gdy ta pozycja jest przydzielana automatycznie to tak naprawdę ma miejsce przypisanie danemu elementowi jakiejś powierzchni.



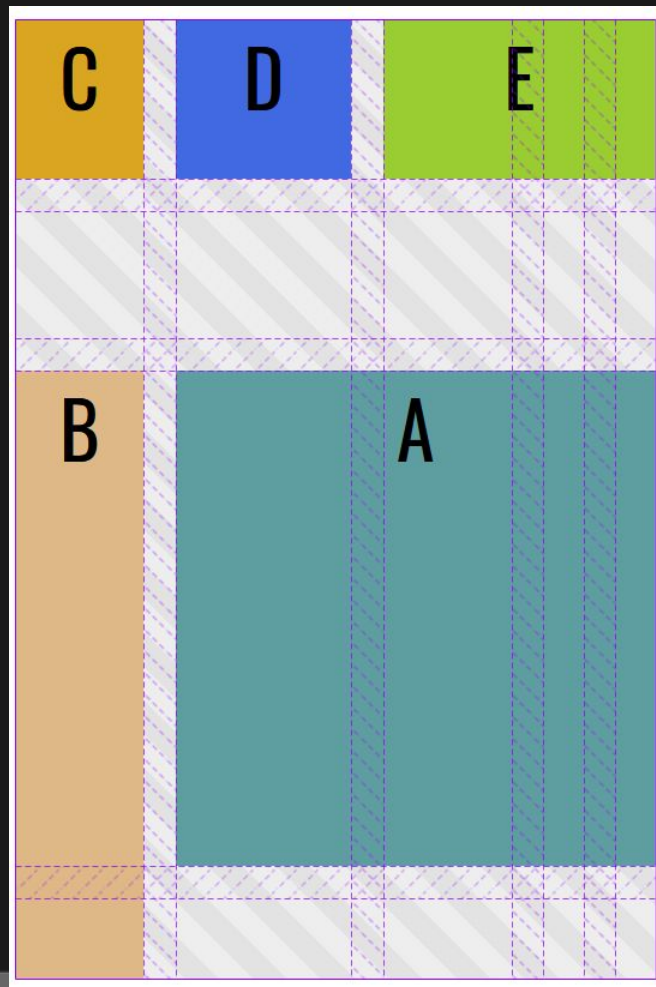


Domyślnie te odstępy (określane jako gutter czy gap) nie występują (mają wielkość 0), ale możemy je ustawić w kontenerze siatki. Możemy nawet określić różne wielkości dla przerw poziomych i pionowych (ale już nie dla przerw między konkretnymi wierszami/kolumnami).



# Gutter/Gap - warto pamiętać

- Nie ma ich na krawędziach kontenera (między krawędzią a kolumną i między krawędzią a wierszem).
- Same nie tworzą kolumn i wierszy.
- Mogą wpływać na wielkość siatki (a przez to na wielkość poszczególnych komórek), ale poza tym nie mają innego wpływu na siatkę.
- Ich zadaniem jest rola marginesów między elementami siatki.
- Jeśli wskażemy, że dany element ma zajmować daną powierzchnię np. od linii 3 kolumny do linii 6 (jak w przykładzie literka E), to zajmuje też powierzchnię przerw między kolumnami. Czy jak w przykładzie z literką B między wierszami.



# Właściwości kontenera

grid-template-columns

grid-template-rows

grid-template-areas

grid-auto-columns

grid-auto-rows

grid-auto-flow

column-gap (grid-column-gap)

row-gap (grid-row-gap)

gap (grid-gap)

justify-items

align-items

place-items

justify-content

align-content

place-content



# Właściwości elementu

grid-column-end

grid-column-start

grid-row-end

grid-row-start

grid-column

grid-row

grid-area

justify-self

align-self

place-self

order

z-index (oczywiście nie tylko CSS Grid, ale tutaj ma swoją rolę)



# Jednostka, funkcje i wartości Grid

fr

repeat()

minmax()

auto-fit

auto-fill


min-content

max-content

span

dense





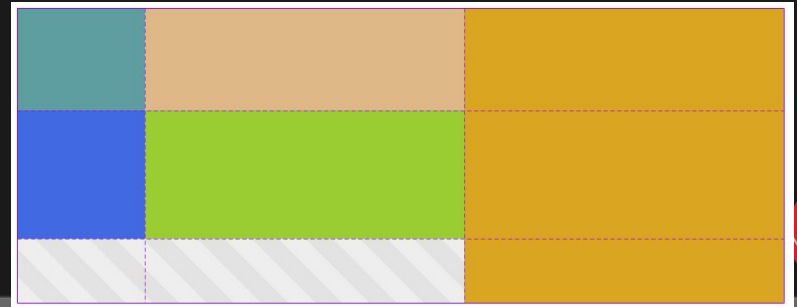
Generowanie struktury siatki  
i umieszczanie w niej elementów.



# Struktura siatki i rozmieszczenie elementów

Sposób 1 - Jawne tworzenie struktury siatki (rozmiar i liczba wierszy i kolumn) oraz jawne umieszczanie elementów w siatce (wskazanie powierzchni siatki, którą ma zajmować dany element).

Sposób 2 - niejawne tj. automatyczne wg. reguł, które można dopasowywać. Elementy siatki są rozkładane automatycznie. W procesie rozmieszczania elementów powstaje też siatka (niejawny sposób powstawania siatki).



# Struktura siatki i rozmieszczenie elementów


Sposób 3 - Wymuszenie struktury siatki poprzez wskazanie pozycji elementu. Wskazanie pozycja elementu (np. od 3 do 4 linii wiersza i od 2 do 4 linii kolumn) wymusi powstanie trzech wierszy i trzech kolumn (jeśli ich wcześniej nie było).



Zaraz zobaczysz na czym te wszystkie sposoby polegają. Warto jednak już teraz wiedzieć, że w praktyce bardzo często te sposoby się mieszają np. jawna siatka, ale automatyczne umieszczanie wszystkich bądź części elementów siatki. Czy jawna siatka, ale część jej struktury powstaje niejawnie.







# Jak tworzona jest struktura siatki



# Siatka jawna (precyzyjna) i niejawna (automatyczna)

CSS Grid tworzy strukturę siatki na trzy sposoby. Możemy to zrobić jawnie, może to być zrobione automatycznie lub może to zostać wymuszone.

- **jawnie** jeśli definiujemy w kontenerze jej strukturę tzn. określamy ile wierszy i kolumn (i o jakiej wielkości) ma. Gdy definiujemy przynajmniej jeden z tych wymiarów.
- **niejawnie (automatycznie)** - elementy siatki mogą tworzyć kolejne kolumny lub wiersze (domyślnie wiersze) o ile siatka nie została zdefiniowana lub została zdefiniowana, ale nie ma możliwości dodania elementu do komórek już istniejących.

```
.grid {  
  display: grid;  
  /* jawnie deklarowana struktura */  
  /* ilość podanych wartości odpowiada  
    liczbie kolumn/wierszy */  
  grid-template-columns: 100px 2fr auto;  
  grid-template-rows: 80px 100px 50px;  
}
```

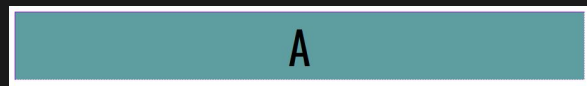


# Wymuszanie struktury siatki

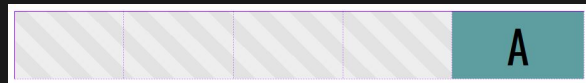
- jawnie (...)
- niejawnie (...)
- przez wymuszenie :) - nadając elementom siatki pozycję w siatce gdy taka pozycja nie istnieje. Załóżmy, że siatka ma tylko jeden wiersz i jedną kolumnę i jeden element siatki. Jeśli dla tego elementu określimy pozycję by był on w 5 kolumnie (po 5 linii), to wymusimy na siatce stworzenie tych pięciu kolumn.

W większości layoutów techniką którą wybierzesz będzie jawne deklarowane siatki w kontenerze (przynajmniej dla kolumn).

AUTOMATYCZNE



WYMUSZENIE



```
/* w elemencie siatki */  
.e1 {  
  grid-column: 5;  
}
```



## Przykłada na strukturze HTML w dalszej części prezentacji

```
<div class="grid">
  <div class="e1">A</div>
  <div class="e2">B</div>
  <div class="e3">C</div>
  <div class="e4">D</div>
  <div class="e5">E</div>
</div>
```

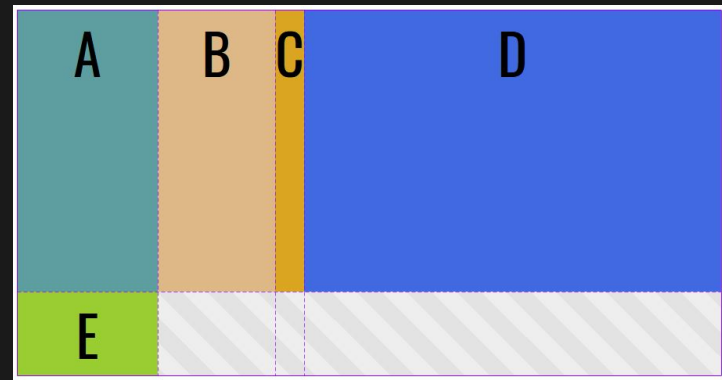
```
.grid {
  display: grid;
}

.e1 {
  text-align: center;
  background-color: cadetblue;
}
```




# grid-template-columns i grid-template-rows

```
.grid {  
  display: grid;  
  grid-template-columns: 20% 100px auto 1fr;  
  grid-template-rows: 5em auto;  
}
```



/\* zadeklarowane 4 kolumny i 2 wiersze (8 komórek co zawsze wynika z pomnożenia jednej wartości przez drugą) \*/  
/\* 5 elementów siatki, domyślnie każdy element zajmuje jedną komórkę (o tym będziemy mówić za chwilę) \*/  
/\* struktura siatki zajmuje całą powierzchnię kontenera, ale nie musi tak być za każdym razem\*/





W jaki sposób umieszczane  
są elementy siatki w siatce



# Element siatki w siatce

W jaki sposób ustalane jest miejsce elementu siatki w siatce?

Pierwszeństwo ma **jawna deklaracja**, gdzie elementowi siatki możemy przypisać daną powierzchnię siatki (komórkę lub większą powierzchnię) - istnieje tutaj kilka techniki, z których możemy skorzystać.

Drugi sposób to **niejawne (automatyczne)** umieszczenie elementów w siatce. Jeśli nie zadeklarujemy gdzie CSS Grid sobie poradzi stosując swoje reguły (które możemy też zmieniać). CSS Grid pozwala ustalić (i ma domyślne) ustawienia na tworzenie struktury i umieszczanie elementów. W skrócie, domyślne ustawienie wygląda tak: Elementy siatki, zajmuje pozycję w wierszu, jeśli nie ma już w nim miejsca (wolnych komórek), przejdź do kolejnego. Jeśli nie ma kolejnego wiersza to zostanie stworzony.



# Automatyczne umieszczanie elementów w strukturze siatki połączone z tworzeniem siatki

```
/* CSS */  
.grid {  
  display: grid;  
}
```

```
/* HTML */  
<div class="grid">  
  <div class="e1">A</div>  
</div>
```

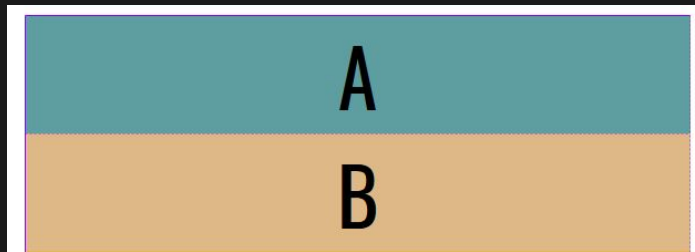




# Automatyczne umieszczanie elementów w strukturze siatki połączone z tworzeniem siatki

```
/* CSS */  
.grid {  
  display: grid;  
}
```

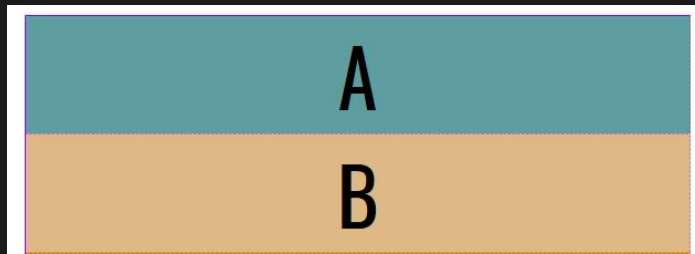
```
/* HTML */  
<div class="grid">  
  <div class="e1">A</div>  
  <div class="e2">B</div>  
</div>
```



# Automatyczne umieszczanie elementów w strukturze siatki połączone z tworzeniem siatki

```
/* CSS */  
.grid {  
  display: grid;  
  /* domyślna - wrócimy do tego*/  
  grid-auto-flow: row;  
}
```

```
/* HTML */  
<div class="grid">  
  <div class="e1">A</div>  
  <div class="e2">B</div>  
</div>
```



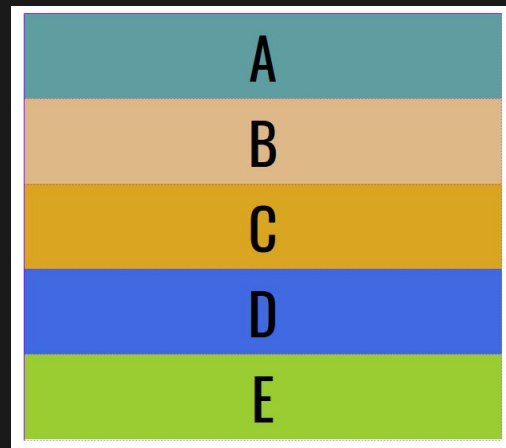
## Automatyczne umieszczanie elementów w strukturze siatki połączone z tworzeniem siatki

```
/* CSS */  
.grid {  
  display: grid;  
}  
/* HTML */  
<div class="grid">  
  <div class="e1">A</div>  
  <div class="e2">B</div>  
  <div class="e3">C</div>  
  <div class="e4">D</div>  
  <div class="e5">E</div>  
</div>
```



# Automatyczna struktura i automatyczne umieszczenie elementów

```
.grid {  
  display: grid;  
  /* tylko deklaracja siatki, bez struktury */  
}
```

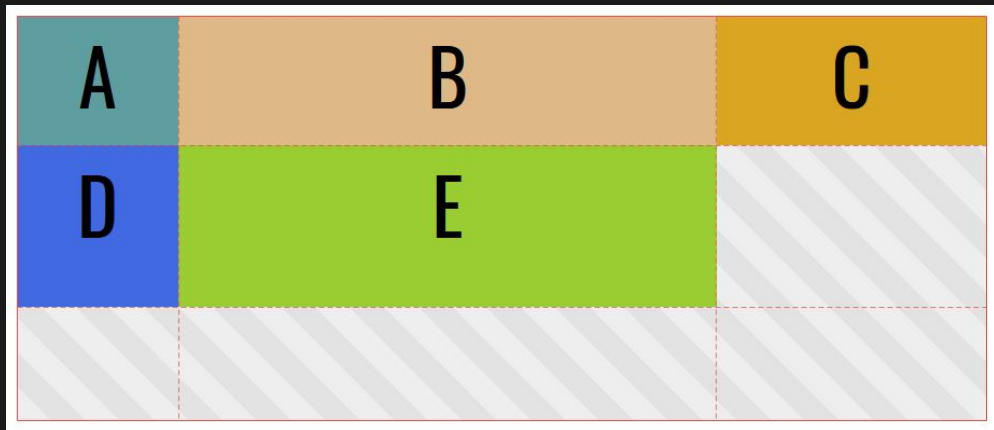


W przykładzie powyżej każdy element siatki (5 elementów) spowodował powstanie nowego wiersza. Wiersz został utworzony bo nie było wolnych przestrzeni w siatce - gdyby były to domyślnie zostałyby uzupełnione komórki jednego wiersza, potem kolejnego i dopiero gdyby ich zabrakło zostałyby stworzony nowy wiersz.



# Automatyczne umieszczanie elementów w istniejącej siatce

```
.grid {  
  display: grid;  
  grid-template-columns: 100px 2fr 1fr;  
  grid-template-rows: 80px 100px 70px;  
}
```



Umieszczenie elementów nastąpiło w kolejności jak w strukturze HTML i po jeden element na komórkę - to podstawowe zasady automatycznego umieszczania elementów w siatce CSS.



# Automatyczne umieszczanie elementów w istniejącej siatce

```
.grid {  
  display: grid;  
  grid-template-columns: 100px 200px;  
  grid-template-rows: 80px 100px;  
}
```

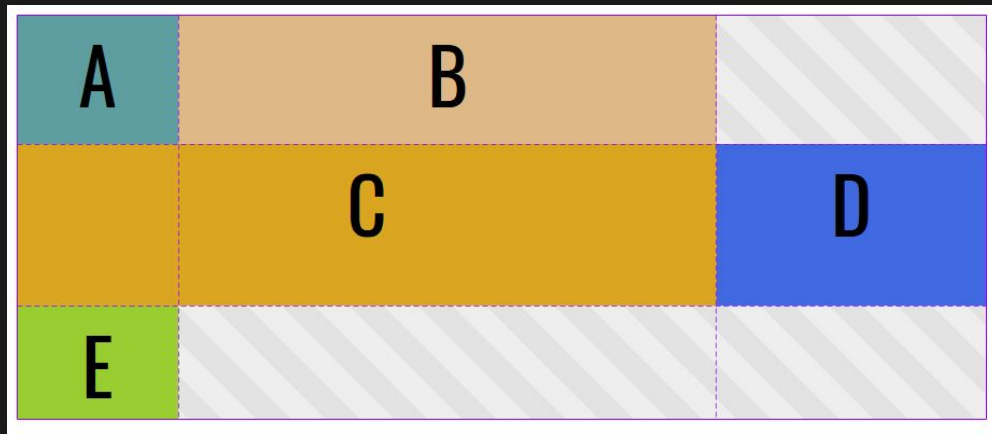


Brakuje miejsca w istniejącej siatce na element(y) siatki? Zostanie utworzony (domyślnie) kolejny wiersz.



# Automatyczne umieszczanie elementów w bardziej skomplikowanej strukturze - dziwne?

```
.grid {  
  display: grid;  
  grid-template-columns: 100px 2fr 1fr;  
  grid-template-rows: 80px 100px 70px;  
}  
  
.e3 {  
  background-color: goldenrod;  
  /* ma zajmować dwie kolumny */  
  grid-column-end: span 2;  
}
```

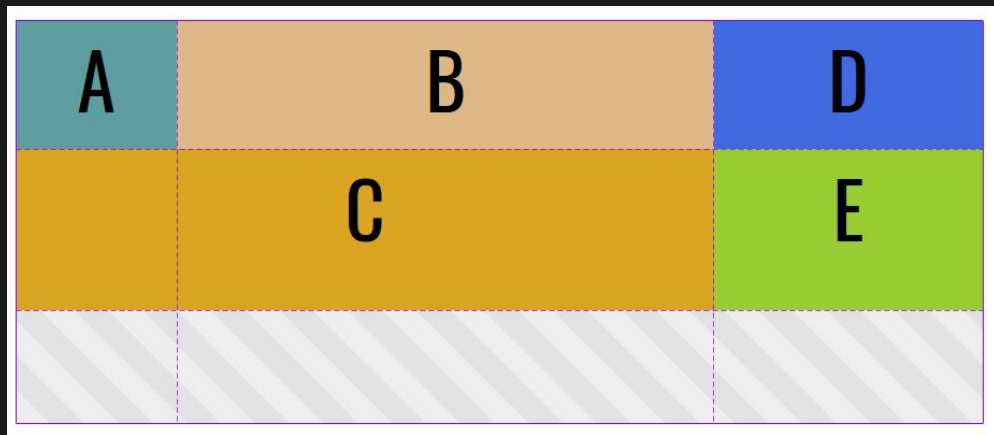


Element D nie zajął wolnego miejsca pozostawionego po elemencie C. Została mu nadana pierwsza wolna komórka, ale nie w ogóle, a po ostatnim elemencie :).



# Automatyczne umieszczanie elementów w bardziej skomplikowanej strukturze

```
.grid {  
  display: grid;  
  grid-template-columns: 100px 2fr 1fr;  
  grid-template-rows: 80px 100px 70px;  
  grid-auto-flow: row dense;  
}  
  
.e3 {  
  background-color: goldenrod;  
  /* ma zajmować dwie kolumny */  
  grid-column-end: span 2;  
}
```



Wartość `dense` pozwala zająć też wolne komórki przed występującymi wcześniej elementami.





# Jawne umieszczanie elementów w istniejącej strukturze

Biorąc to pod uwagę, może lepiej jest jawnie zadeklarować strukturę siatki i pozycję elementów ;) Czasami coś sztywnego to duża zaleta.

Na przykład coś takiego:

(kontener) 3 kolumny o takich i takich wielkościach. 4 wiersze. pierwszy 100px, drugi się rozciąga, 3 i 4 tyle ile potrzebują.

(elementy siatki) Ty elemencie w tej komórce. Ty w kolumnie 2 i 3. Ty od 3 wiersza do ostatniego.

A ty we wszystkich komórkach o nazwie “footer”.

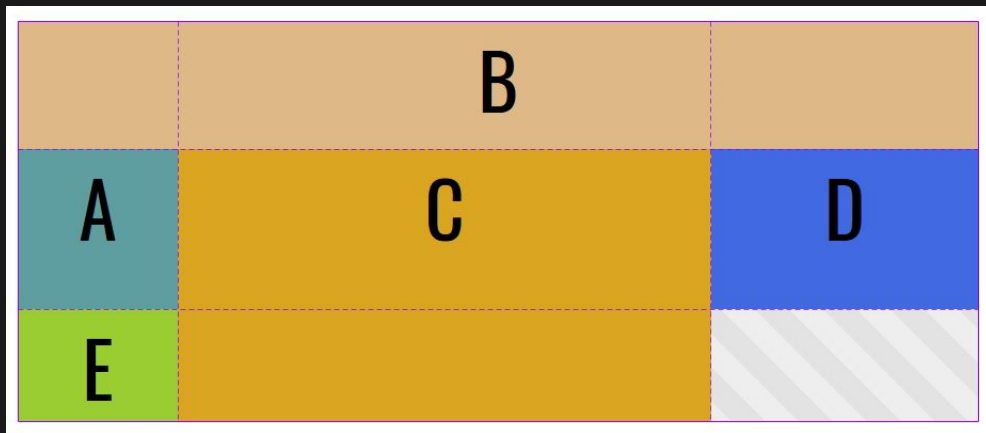


# Jawne umieszczanie części elementów

```
.grid {  
  display: grid;  
  grid-template-columns: 100px 2fr 1fr;  
  grid-template-rows: 80px 100px 70px;  
}
```

```
.e2 {  
  grid-row: 1;  
  grid-column: 1/-1  
}
```

```
.e3 {  
  grid-column: 2;  
  grid-row-start: 2;  
  grid-row-end: span 2;  
}
```



Elementy z jawną deklaracją pozycji zajmują swoje miejsce. Dopiero wtedy elementy bez jawnego określenia pozycji są rozstawiana automatycznie.



Możesz mieć mętlik w głowie

To dobrze, trudno się w kilkadziesiąt minut z tak skomplikowaną technologią poznać i polubić ;)

Zaraz na przykładach będzie łatwiej... i trudniej bo jest mnóstwo właściwości, wartości i funkcji.





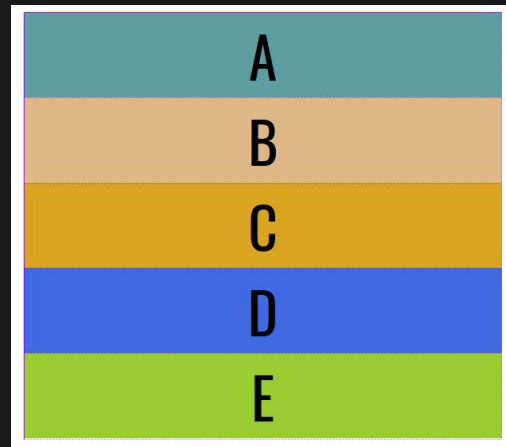
# Zabawa z przykładami

- tworzenie struktury i rozmieszczenia automatyczne



## Brak miejsca - nowa kolumna czy wiersz?

```
.grid {  
  display: grid;  
}
```



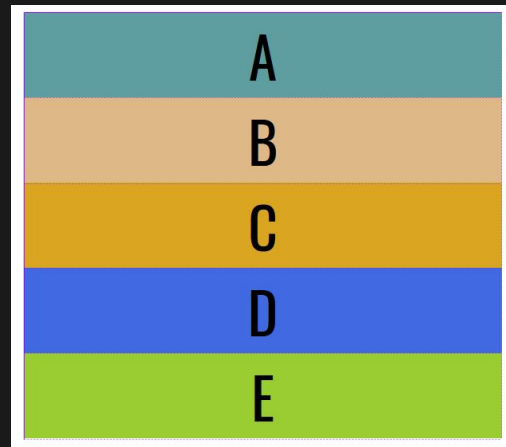
To już wiesz: Ustanowienie kontenera siatki, bez jawnej deklaracji struktury siatki, oznacza, że struktura dla elementów siatki tworzona jest automatycznie. Element siatki, o ile nie będzie dla niego wolnej komórki, będzie więc tworzyć nową kolumnę\* lub wiersz\* (domyślnie) - w ten sposób dla elementu siatki powstanie nowa przestrzeń, którą będzie ten element zajmował.

\* właściwość, która za to odpowiada to [grid-auto-flow](#), ustawiona domyślnie na [row](#) - wrócimy do niej za chwilę.



# Wielkości niejawnych kolumn i wierszy

```
.grid {  
  display: grid;  
  // grid-auto-rows: auto;  
  // grid-auto-columns: auto;  
}
```

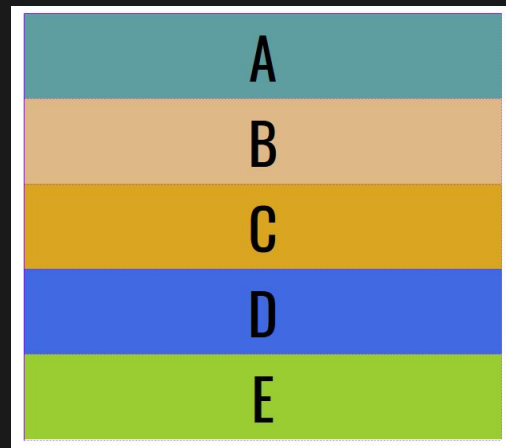


Widoczna szerokość kolumny wynosi tutaj 100 proc. kontenera - co samo w sobie może być dziwne ponieważ dwie właściwości odpowiedzialne za wielkość powstających automatycznie kolumn i wierszy (są to `grid-auto-rows` i `grid-auto-columns`) ustawione są na `auto`, a `auto` to w skrócie taka wielkość jaka jest potrzebna na zawartość. Jednak za takie “rozciąganie” kolumna i (ewentualnie) wierszy odpowiadają inne właściwości `justify-content` i `align-content`.



# Wielkości niejawnych kolumn i wierszy

```
.grid {  
  display: grid;  
  // grid-auto-rows: auto;  
  // grid-auto-columns: auto;  
  // grid-auto-flow: row;  
}
```



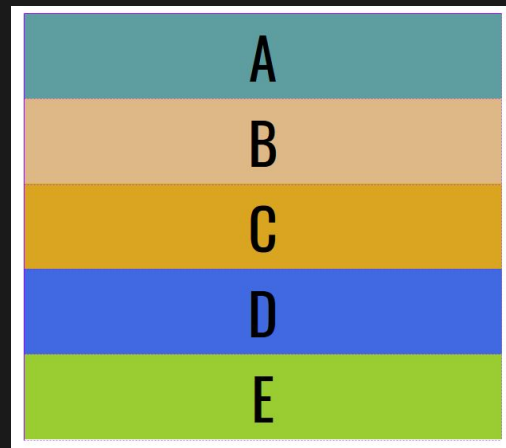
Natomiast jeśli chodzi o każdy nowy wiersz, to powiększa on domyślnie kontener siatki (jeśli jego wysokość nie jest określona) o wysokość której potrzebuje (jeśli wysokość kontenera nie jest określona). W tym wypadku, który widzimy, wiersz ma taką wielkość by zmieścić zawartość, która jest w nim (literę z interlinią).

Jeśli brakuje miejsca dla elementu, to oznacza to (domyślnie) stworzenie nowego wiersza. Wynika to z domyślnego ustawienia właściwości `grid-auto-flow` (wskazującego na `row`).



# Kolejność i jedna komórka na element

```
.grid {  
  display: grid;  
}
```



To już wiesz, ale powiedzmy to ostatni raz.

- Elementy siatki w procesie automatycznego (niejawnego) przypisania im powierzchni siatki zajmują powierzchnię tylko jednej komórki. Przydzielenie powierzchni następuje zgodnie z kolejnością zajmowaną przez dany element w strukturze HTML i zgodnie z właściwością `grid-auto-flow`.
- jeśli zdecydujemy się na jawne wskazanie powierzchni dla danego elementu siatki, to takie deklarowanie miejsca ma pierwszeństwo. Nadal jednak elementy, które nie mają jawnego przypisania pozycji będą układać się automatycznie i mogą wpływać na strukturę siatki.



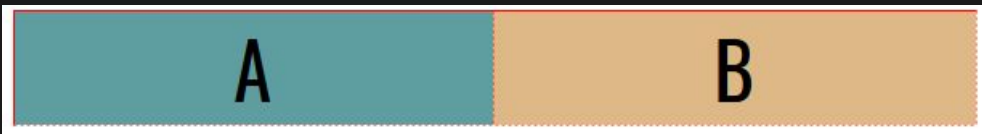


# Zmiany w automatycznym tworzeniu siatki

```
.grid {  
  display: grid;  
  grid-auto-flow: column; /* domyślnie row */  
}
```

Gdy zamiast row użyjemy wartości column to:

- CSS Grid będzie chciał w pierwszej kolejności zappełnić całą kolumnę a następnie kolejne wolne kolumny.
- W przypadku braku wolnej powierzchni (komórki) w siatce stworzona zostanie nowa kolumna, a nie wiersz (co widać na przykładzie poniżej)

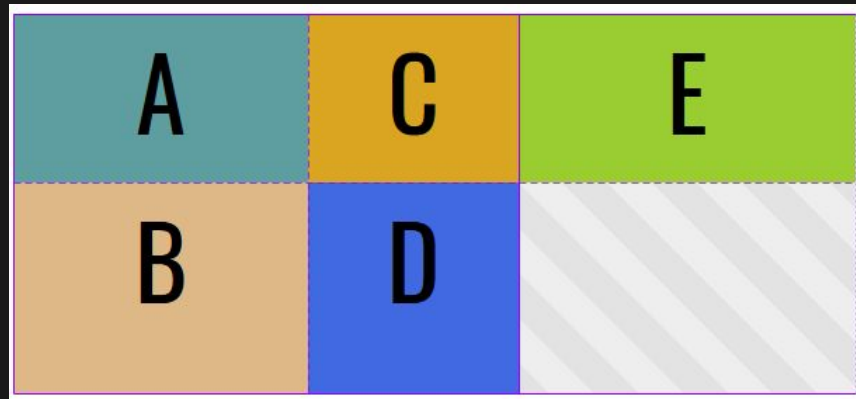


Przykład zachowania siatki gdy przy tych samych ustawieniach dodamy do niej 2 elementy i 5 elementów.



# grid-auto-flow ustawione na column

```
.grid {  
  display: grid;  
  grid-auto-flow: column;  
  width: 400px;  
  grid-template-columns: 140px 100px;  
  grid-template-rows: 80px 100px;  
}
```

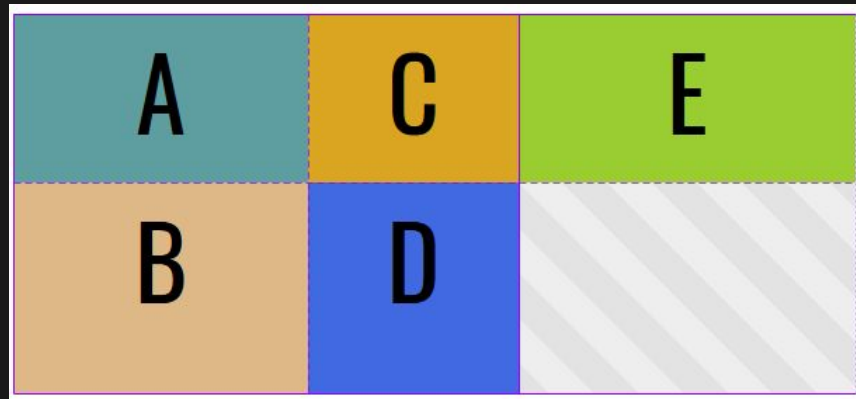


- zwróć uwagę na sposób rozmieszczenia, najpierw kolumna 1, potem druga. Ponieważ brakuje miejsca dla piątego elementu niejawnie zostaje utworzona trzecia kolumna.
- zwróć uwagę na wielkość ostatniej (automatycznie utworzonej) kolumny. Jest on rozciągnięty na całą pozostałą wielkość kontenera. Przypomnijmy, że odpowiada za to właściwość grid-auto-column ustawiona na auto oraz właściwość justify-content (która domyślnie rozciąga komórkę na wolną przestrzeń jeśli ta wolna przestrzeń istnieje).



# grid-auto-flow ustawione na column

```
.grid {  
  display: grid;  
  grid-auto-flow: column;  
  width: 400px;  
  grid-template-columns: 140px 100px auto;  
  grid-template-rows: 80px 100px;  
}
```

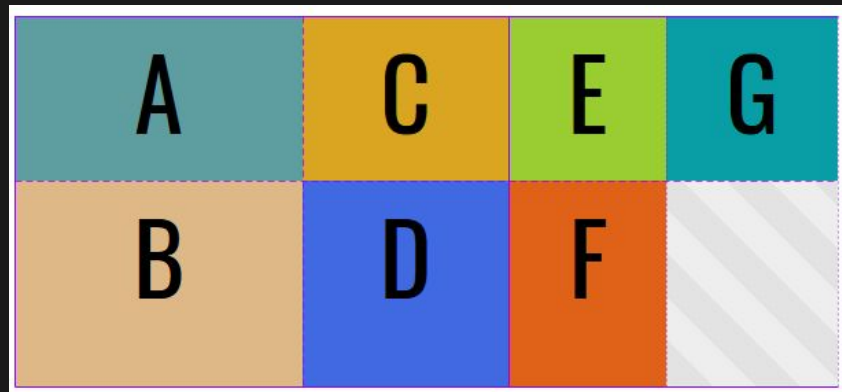


Każda nowa, automatyczna kolumna w istocie zachowa się tak samo jak w przypadku dodania wartości auto do grid-template-columns. Przy kolejnym wierszu byłoby to 140px 100px auto auto. Sama wartość nowej kolumny jest pobierana z właściwości grid-auto-column (domyślnie auto).



# Rosnąca siatka

```
.grid {  
  display: grid;  
  grid-auto-flow: column;  
  width: 400px;  
  grid-template-columns: 140px 100px;  
  grid-template-rows: 80px 100px;  
}
```

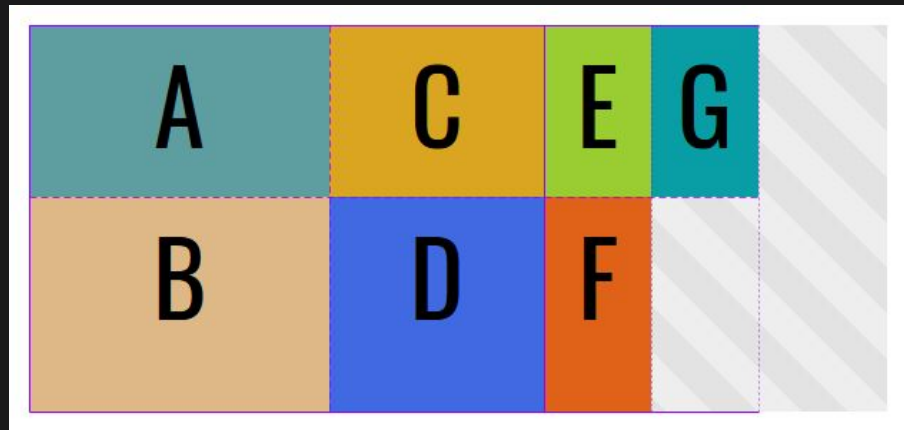


- dodane zostały dwa kolejne elementy. Pierwszy zajął wolną komórkę w kolumnie, a drugi stworzył kolejną kolumnę
- 3 i 4 kolumna zostały stworzone automatycznie. Za to jaką mają wielkość odpowiada grid-auto-columns, domyślnie ustawione na auto - to już wiesz. Zaraz to zmienimy.



# Zmiany w automatycznym tworzeniu siatki

```
.grid {  
  display: grid;  
  grid-auto-flow: column;  
  width: 400px;  
  grid-template-columns: 140px 100px;  
  grid-template-rows: 80px 100px;  
  grid-auto-columns: 50px; /*domyślnie auto */  
}
```



- nowo powstające kolumny mają po 50px
- dokładnie tak samo zachowa się w tym momencie siatka jeśli napiszemy  
`grid-template-columns: 140px 100px 50px 50px;`



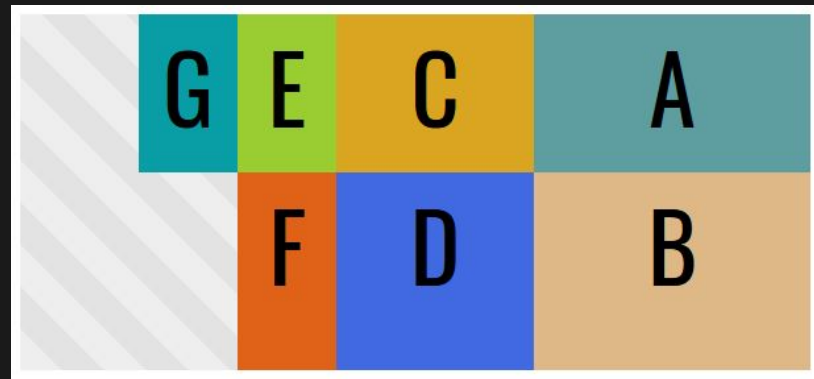
# Zmiana kierunku

```
.grid {  
  display: grid;  
  grid-auto-flow: column;  
  direction: rtl;  
  /* domyślnie ltl - pamiętajmy, że wpływa to nie tylko na CSS Grid, więc raczej z tego  
  oczywiście nie skorzystamy, chyba, że robimy projekt np. dla krajów arabskich */  
}
```



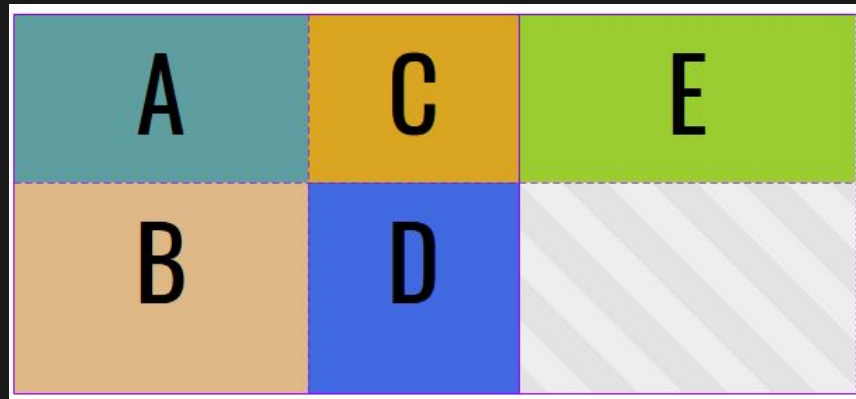
# Zmiany w automatycznym tworzeniu siatki

```
.grid {  
  display: grid;  
  grid-auto-flow: column;  
  width: 400px;  
  grid-template-columns: 140px 100px;  
  grid-template-rows: 80px 100px;  
  grid-auto-columns: 50px; /*domyślnie auto */  
  direction: rtl;  
}
```



# Zmiany w automatycznym tworzeniu siatki

```
.grid {  
  display: grid;  
  grid-auto-flow: column;  
  width: 400px;  
  grid-template-columns: 140px 100px;  
  grid-template-rows: 80px 100px;  
}
```



Wróćmy do “normalnego” przykładu. I zmienimy `grid-auto-flow` z `column` (ten slajd) na `row` (następny slajd) - przypomnę, że `row` jest wartością domyślną, więc gdy nie deklarujemy tej wartości domyślnie jest właśnie `row`.



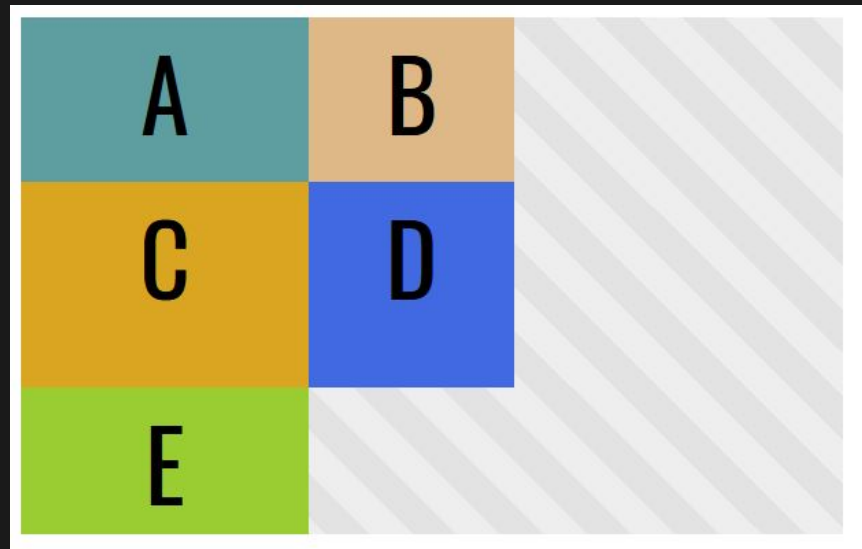


# Zmiany w automatycznym tworzeniu siatki

```
.grid {  
  display: grid;  
  grid-auto-flow: row;  
  width: 400px;  
  grid-template-columns: 140px 100px;  
  grid-template-rows: 80px 100px;  
}
```

Ależ zmiana! Czy to jest normalne?? Tak, jest ok :)

- zapełnia całą wiersz (wiersz nie musi zajmować całego kontenera)
- przechodzi potem do kolejnego wiersza
- gdy brakuje wierszy tworzy nowy wiersz

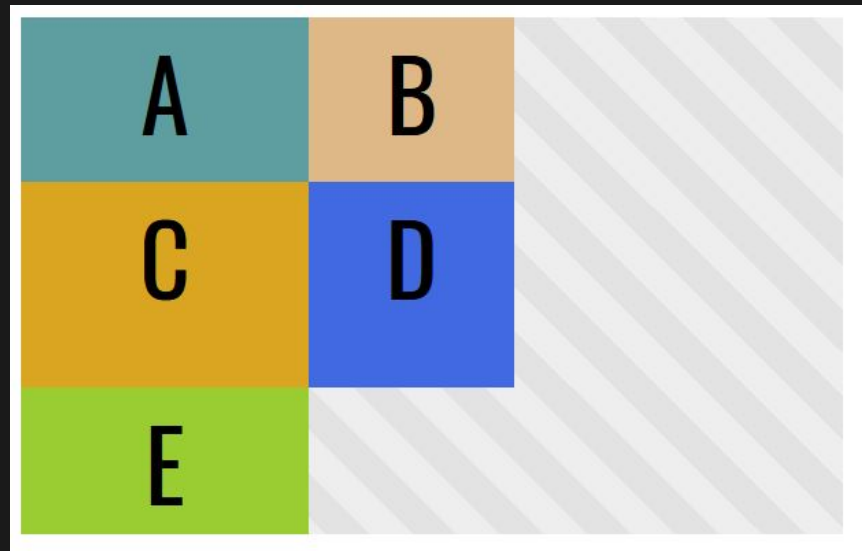


# Zmiany w automatycznym tworzeniu siatki

```
.grid {  
  display: grid;  
  grid-auto-flow: row;  
  width: 400px;  
  grid-template-columns: 140px 100px;  
  grid-template-rows: 80px 100px;  
  // grid-auto-rows: auto;  
}
```

Każdy nowy (tworzony niejawnie) wiersz ma domyślnie wielkość auto, co wynika z właściwości `grid-auto-rows` ustawionej na auto.

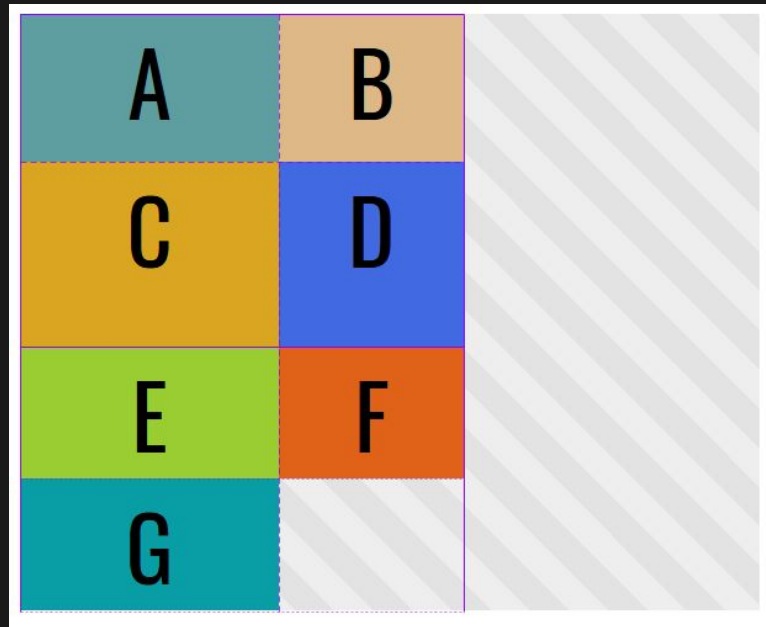
Gdy brakuje przestrzeni na element siatki, tworzony jest nowy wiersz



# Zmiany w automatycznym tworzeniu siatki

```
.grid {  
  display: grid;  
  grid-auto-flow: row;  
  width: 400px;  
  grid-template-columns: 140px 100px;  
  grid-template-rows: 80px 100px;  
}
```

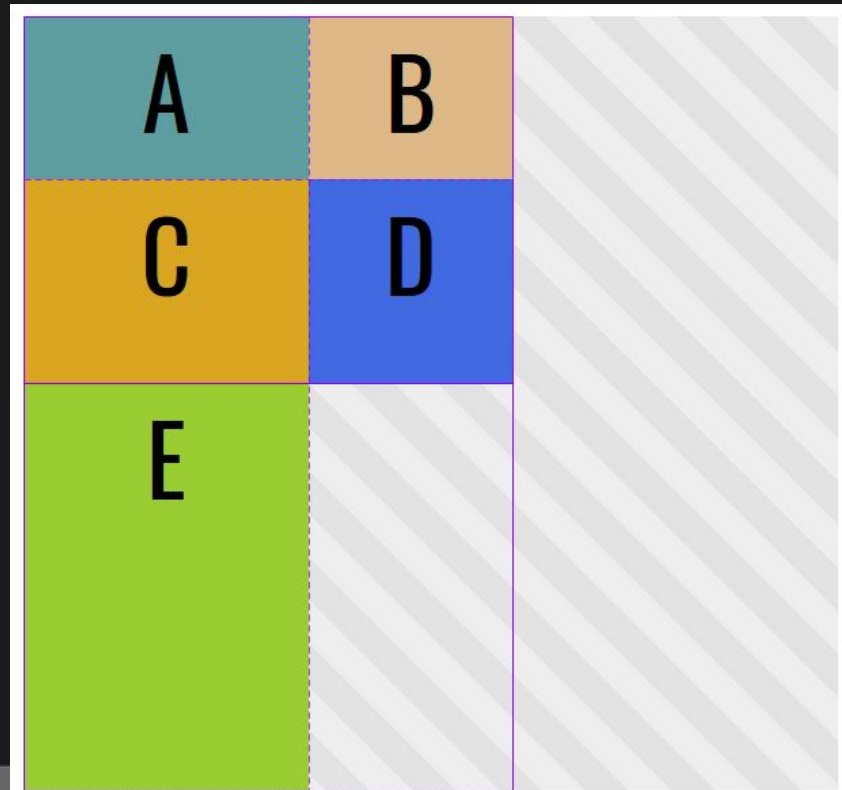
Dodajmy jeszcze 2 elementy siatki (dwa kolejne elementy HTML) i zobaczmy efekt. Wiersz 3 został wypełniony. Ponieważ brakuje miejsca na element 7 (G) stworzony zostanie kolejny wiersz. Dodane wiersze mają tę samą wielkość (bo takiej samej wielkości potrzebują, więc auto daje w tym wypadku ten sam efekt).



# Zmiany w automatycznym tworzeniu siatki

```
.grid {  
  display: grid;  
  grid-auto-flow: row;  
  width: 400px;  
  grid-template-columns: 140px 100px;  
  grid-template-rows: 80px 100px;  
  grid-auto-rows: 200px; // domyślnie auto  
}
```

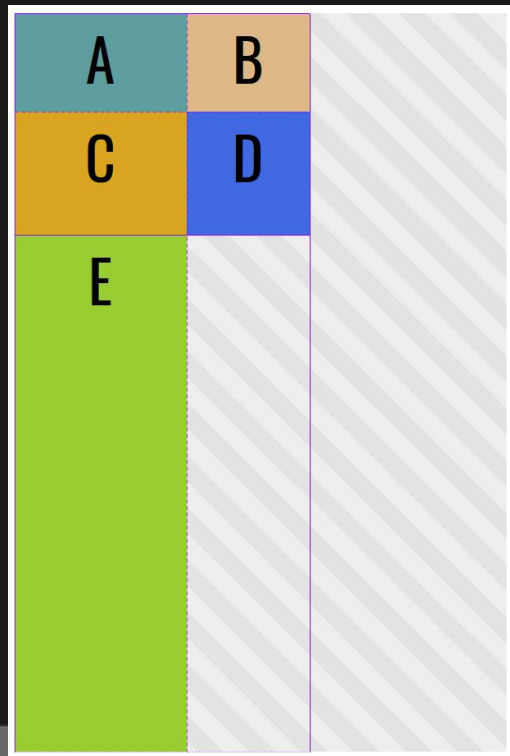
Wróćmy do 5 elementów, ale zmieńmy wielkość automatycznie tworzonego wiersza z auto na 200px.



# Zmiany w automatycznym tworzeniu siatki

```
.grid {  
  display: grid;  
  grid-auto-flow: row;  
  width: 400px;  
  height: 600px;  
  grid-template-columns: 140px 100px;  
  grid-template-rows: 80px 100px;  
  grid-auto-rows: auto;  
}
```

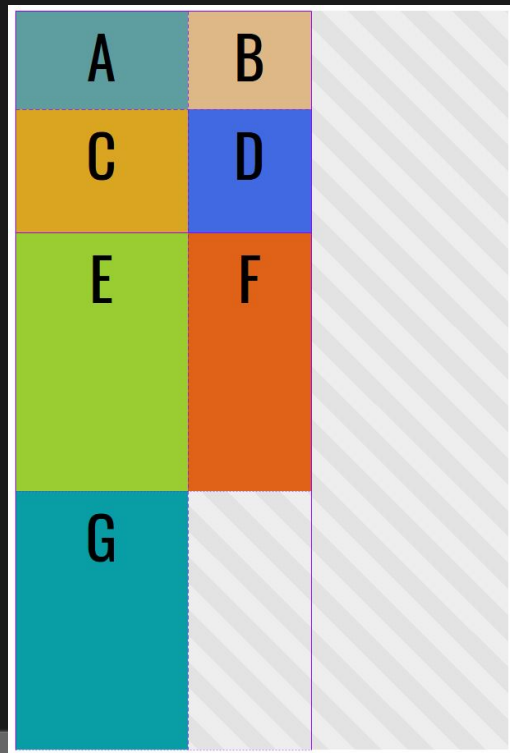
Dla naszego przykładu powiększymy teraz wysokość kontener do 600px i przywracamy grid-auto-rows na auto. Nowy wiersz ma wartość auto, jest jednak rozciągany (podobnie jak kolumny), co wynika z ustawień właściwości justify-content (kolumny) i align-content (wiersz) na stretch.



# Zmiany w automatycznym tworzeniu siatki

```
.grid {  
  display: grid;  
  grid-auto-flow: row;  
  width: 400px;  
  height: 600px;  
  grid-template-columns: 140px 100px;  
  grid-template-rows: 80px 100px;  
  grid-auto-rows: auto;  
}
```

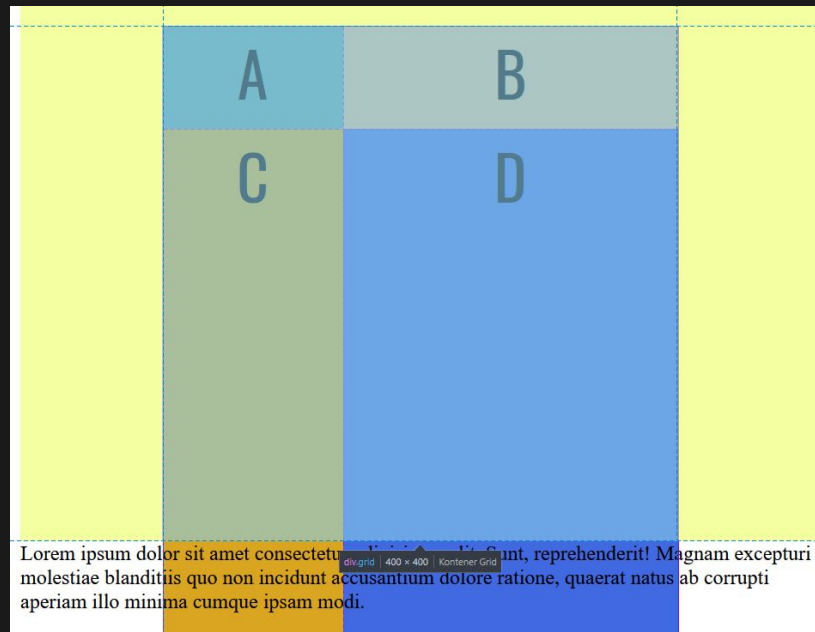
Oczywiście kolejne automatyczne wiersze sprawiają, że przestrzeń do rozdzielenia będzie mniejsza (bo wysokość kontenera określiliśmy sztywno na 600px).



# Zmiany w automatycznym tworzeniu siatki

```
.grid {  
  display: grid;  
  margin: 50px auto 0;  
  width: 400px;  
  height: 400px;  
  grid-template-columns: 140px 260px;  
  grid-template-rows: 80px 500px;  
}  
  
<div class="grid"><!-- ... --></div>  
<section>Lorem ipsum ...</section>
```

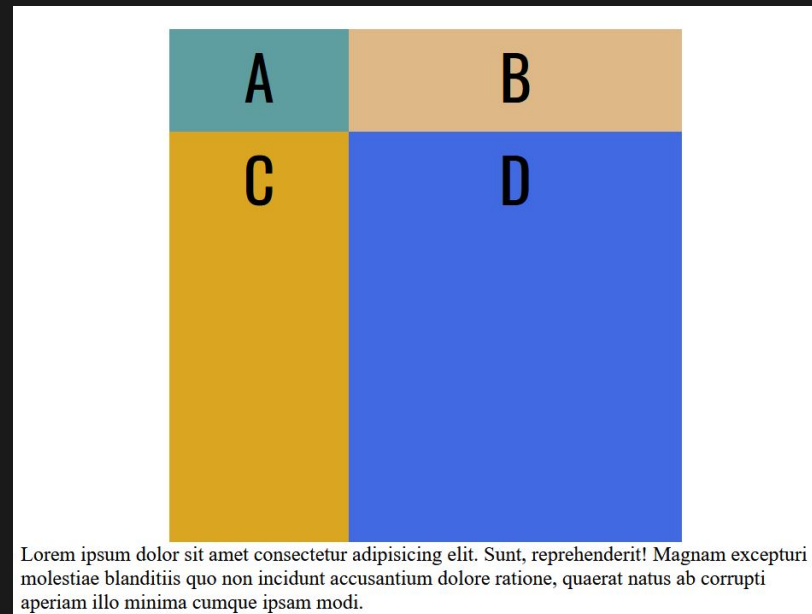
Oczywiście kolejne automatyczne wiersze sprawiają, że przestrzeń do rozdzielenia będzie mniejsza (bo wysokość kontenera określiliśmy sztywno na 600px).



# Zmiany w automatycznym tworzeniu siatki

```
.grid {  
  display: grid;  
  margin: 50px auto 0;  
  width: 400px;  
  height: 400px;  
  grid-template-columns: 140px 260px;  
  grid-template-rows: 80px 500px;  
  overflow: hidden;  
}  
  
<div class="grid"><!-- ... --></div>  
<section>Lorem ipsum ...</section>
```

Można zastosować overflow:hidden;







# Zabawa z przykładami

- rozmieszczenie elementów a siatka automatyczna



# Wskazanie pozycji elementu siatki

```
.grid {  
  display: grid;  
}  
//Element A  
.e1 {  
  grid-column: 3;  
  grid-row: 2/3;  
}
```



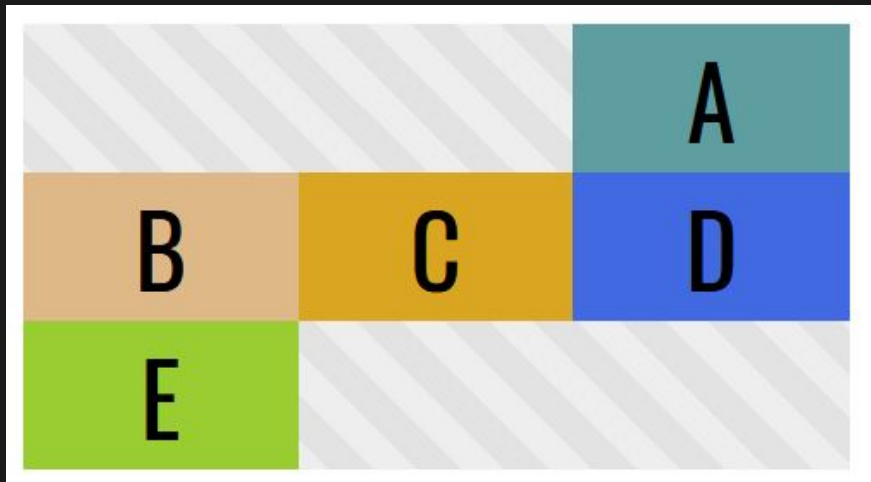
Określenie pozycji powoduje, że dany element nie jest już umieszczany automatycznie (niejawnie).

Na podstawie wskazania pozycji określona jest powierzchnia (pozycja powierzchni) na której umieszczony będzie element.



# Wskazanie pozycji elementu siatki - tylko jeden wymiar

```
.grid {  
  display: grid;  
}  
//Element A  
.e1 {  
  grid-column: 3;  
  // grid-row: 2/3;  
}
```

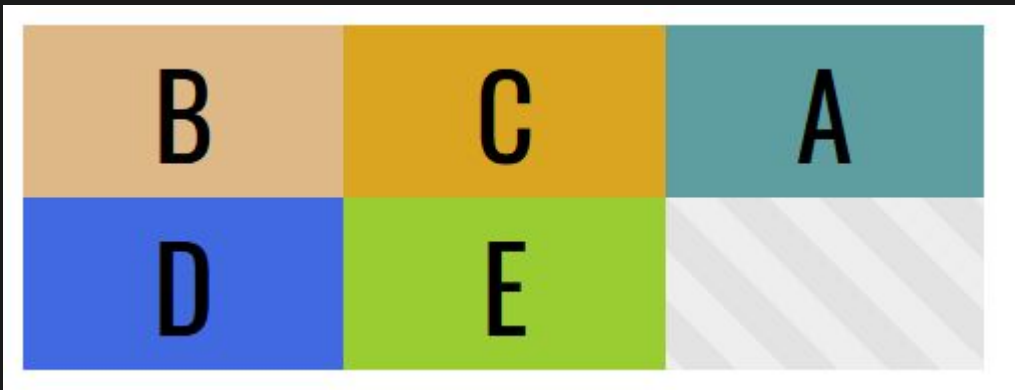


Nie ma znaczenia, czy najpierw określimy pozycję w kolumnie czy wierszu. Nie musimy też zawsze określać obu ścieżek. Jednak jeśli określimy tylko jeden tor, to na drugim będzie wciąż będzie umiejscowiony automatycznie. A to ma swoje konsekwencje, bp takie, że domyślnie w wierszu przed nim nie będą inne elementy.



## Wskazanie pozycji elementu siatki - dwa wymiary

```
.grid {  
  display: grid;  
}  
//Element A  
.e1 {  
  grid-column: 3;  
  grid-row: 1;  
}
```



Często więc zdecydowanie lepiej określić oba wymiary. Wtedy rozmieszczenie jawne (dla każdego wymiaru) ma pierwszeństwo przed tym niejawnym.

Coraz bardziej staram Ci się pokazywać, że jawne deklaracje (zarówno struktury siatki jak i pozycji elementów) mają mnóstwo plusów ;)



# Wskazanie pozycji elementu siatki - wersje skrócone

```
.grid {  
  display: grid;  
}  
//Element A  
.e1 {  
  grid-column: 3; // wersja skrócona.  
  // W praktyce:  
  // grid-column-start: 3; //linia  
  // grid-column-end: auto; // o jedną linie  
  grid-row: 2/3; // wersja skrócona.  
  // W praktyce:  
  // grid-row-start: 2;  
  // grid-row-end: 3;  
}
```



# Wskazanie pozycji elementu siatki - alternatywy

```
.grid {  
  display: grid;  
}  
//Element A  
.e1 {  
  grid-column: 3/4; // wersja skrócona.  
  // W praktyce:  
  // grid-column-start: 3;  
  // grid-column-end: 4;  
  grid-row: 2; // wersja skrócona.  
  // W praktyce:  
  // grid-row-start: 2;  
  // grid-row-end: auto;  
}
```



# Wskazanie pozycji elementu siatki - alternatywy

```
.grid {  
  display: grid;  
}  
//Element A  
.e1 {  
  grid-column: 3/4; // TO SAMO CO SAMO 3 czy 3/auto  
    // W praktyce:  
    // grid-column-start: 3;  
    // grid-column-end: 4;  
  grid-row: 2/3; // TO SAMO CO 2 czy 2/auto.  
    // W praktyce:  
    // grid-row-start: 2;  
    // grid-row-end: auto;  
}
```



# Wskazanie pozycji elementu siatki - span

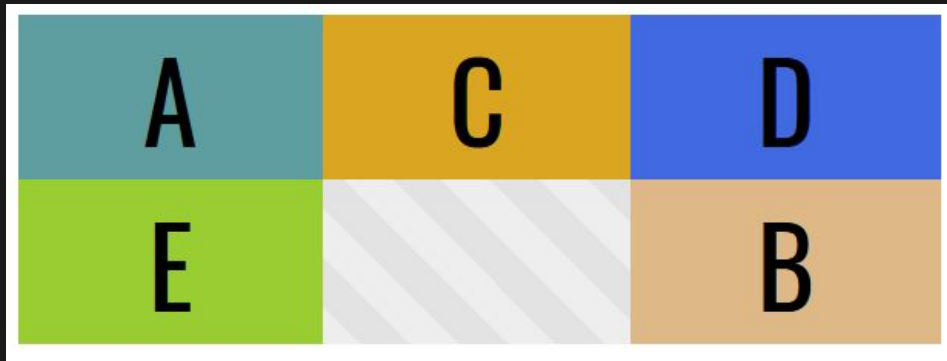
```
.grid {  
  display: grid;  
}  
//Element A  
.e1 {  
  grid-column: 3/span 1; // TO SAMO CO SAMO 3 , 3/4 , 3/auto  
    // W praktyce:  
    // grid-column-start: 3;  
    // grid-column-end: span 1;  
  grid-row: 2/span 1; // TO SAMO CO 2, 2/3 , 2/auto.  
    // W praktyce:  
    // grid-row-start: 2;  
    // grid-row-end: span 1;  
}
```





# Właściwości -start i -end

```
.grid {  
  display: grid;  
}  
  
/* Element B */  
.e2 {  
  grid-column-start: 3;  
    // druga wartość domyślna:  
    // grid-column-end: auto;  
  grid-row-end: 3;  
    // druga wartość domyślna:  
    // grid-row-start: auto;  
}
```



# Wielkość auto i span

```
.grid {  
  display: grid;  
}  
  
/* B */  
.e2 {  
  //dla czystości możemy określić wszystkie wartości  
  grid-column-start: 3;  
  grid-column-end: 4;  
  grid-row-start: 1;  
  grid-row-end: 2;  
}
```



# grid-column i grid-row

```
.grid {  
  display: grid;  
}
```

```
/* B */
```

```
.e2 {  
  //te 4 wartości możemy napisać w dwóch właściwościach  
  grid-column: 3/4;  
  grid-row: 1/2;  
}
```



## grid-area - skrót

```
.grid {  
  display: grid;  
}
```

```
/* B */  
.e2 {  
  //a nawet w jednej zbiorczej właściwości  
  grid-area: 2/3/3/4;  
  // grid-row-start/grid-column-start/grid-row-end/grid-column-end  
}
```



# Wielkość auto - pamiętajmy

```
.grid {  
  display: grid;  
  // grid-auto-rows: auto;  
  // grid-auto-columns: auto;  
}
```

```
/* A */  
.e1 {  
  grid-column: 3;  
  grid-row: 2/4;  
}
```



Mamy 3 wiersze, chociaż trzeciego nie widać (ale widać 4 linie)

Przeglądarka wygeneruje taką wartość: `grid-template-rows: 70px 70px 0;`

70px bo akurat tyle ma ten kontent (interlinia)  
Ale dlaczego, trzecia ma 0? Auto.

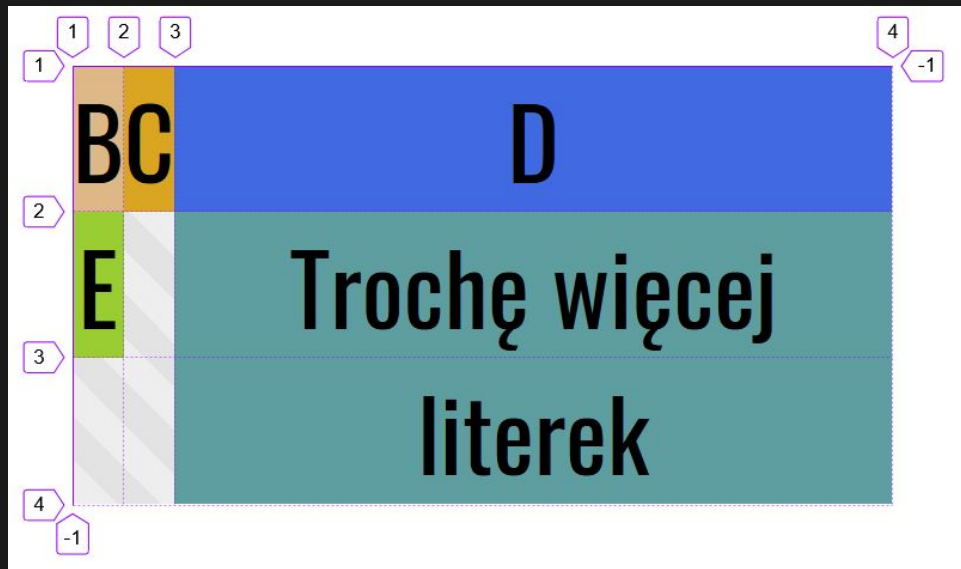
Auto - tyle ile potrzebuje, ale nie mniej niż potrzebuje. W tym wypadku, po prostu nic nie "przechodzi" do kolejnej komórki, którą zajmuje element A.



# Wielkość auto - zachowanie przy różnej zawartości

```
.grid {  
  display: grid;  
  // grid-auto-rows: auto;  
  // grid-auto-columns: auto;  
}
```

```
/* A */  
.e1 {  
  grid-column: 3;  
  grid-row: 2/4;  
}
```



Ponieważ wszystkie tory mają domyślnie auto, dopasowują się do wielkości kontentu.



$$/* \quad A \quad */$$

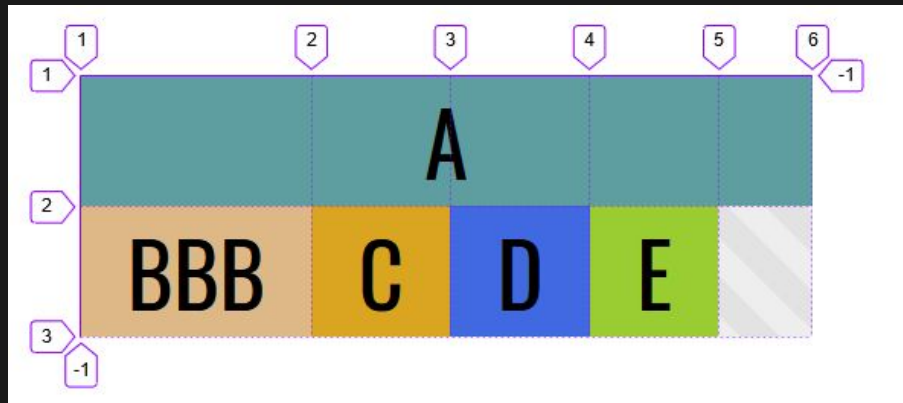
Diagram illustrating a 4x4 grid structure with row and column indices:

- Row indices (left): 1, 2, 3, -1
- Column indices (top): 1, 2, 3, -1
- Grid content:
  - Top-left (orange): BC
  - Top-right (blue): D
  - Bottom-left (green): E
  - Bottom-right (teal): Trochę więcej literek i cyferek (432043204)



# Wielkość auto i span

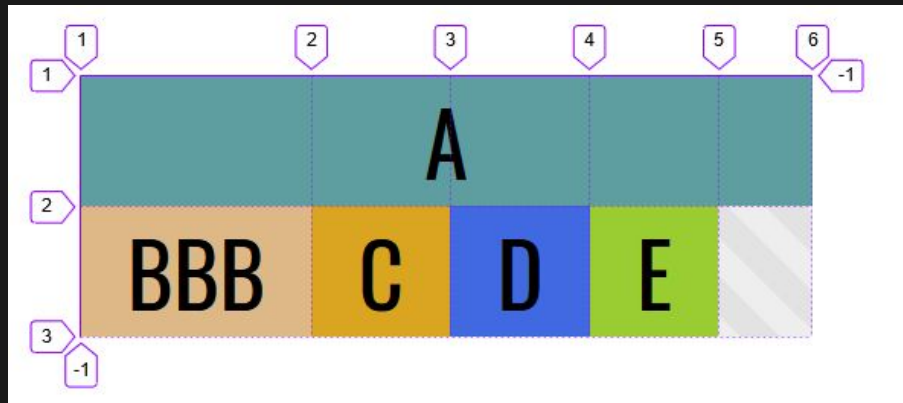
```
.grid {  
  display: grid;  
}  
  
/* A */  
.e1 {  
  grid-column: 1/span 5;  
  // od 1 linii w 5 komórkach (o 5 linii)  
  // grid-column-start: 1;  
  // grid-column-end: span 5;  
  grid-row: 1  
  // grid-row-start: 1;  
  // grid-row-end: auto;  
}
```





# Wielkość auto i span

```
.grid {  
  display: grid;  
}  
  
/* A */  
.e1 {  
  grid-column: 1/6;  
  // od linii 1 do linii 6  
  // grid-column-start: 1;  
  // grid-column-end: 6;  
  grid-row: 1  
  // grid-row-start: 1;  
  // grid-row-end: auto;  
}
```





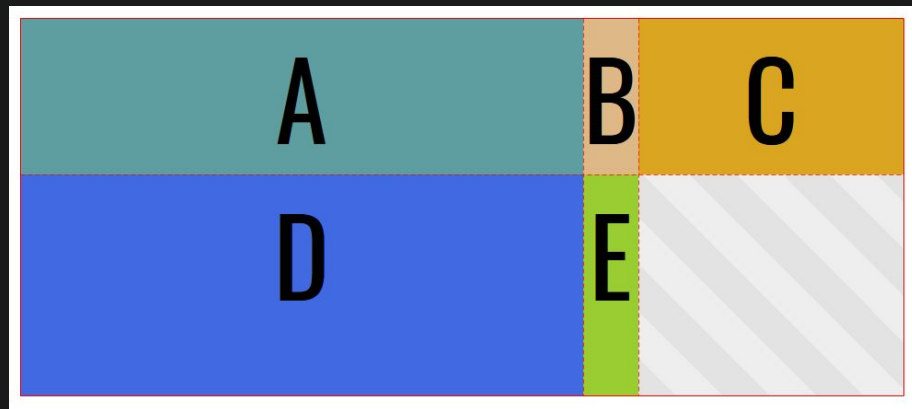
# Zabawa z przykładami

- jawna deklaracja siatki



# Kolumny w siatce

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr auto 30%;  
  grid-template-rows: 1fr 100px;  
}
```



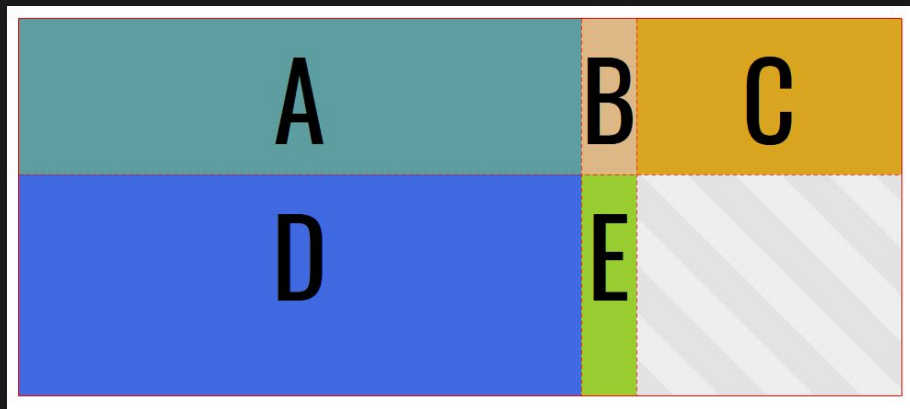
Za pomocą właściwości `grid-template-columns` i `grid-template-rows` możemy w sposób jawny zadeklarować strukturę siatki czyli określić liczbę wierszy i kolumn oraz ich wielkość.

Każda wartość użyta w tych właściwościach oznacza jedną kolumnę (`grid-template-columns`) lub jeden wiersz (`grid-template-rows`)



# fraction (fr)

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr auto 30%;  
  grid-template-rows: 1fr 100px;  
}
```

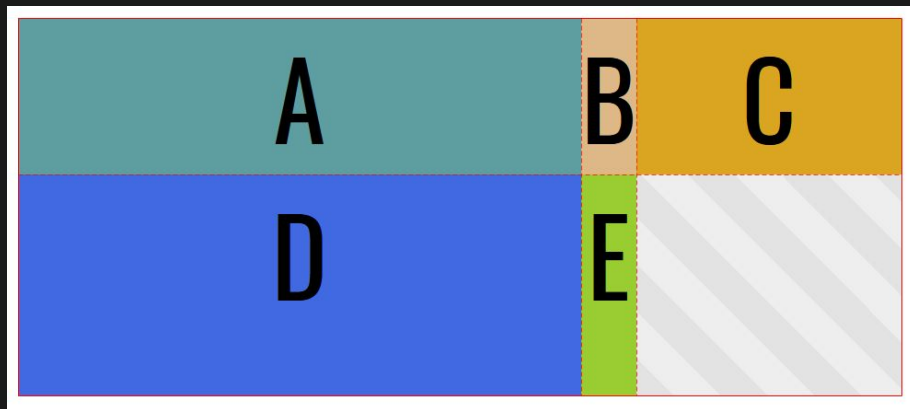


fr - jednostka oznaczająca frakcję (część) - dotyczy wolnej przestrzeni kontenera. Wyrażona w wartościach dodatnich. Frakcja może być wielokrotnie używana jako wartość kolumny czy wiersza.



# Kolumny w siatce

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr auto 30%;  
  grid-template-rows: 1fr 100px;  
}
```

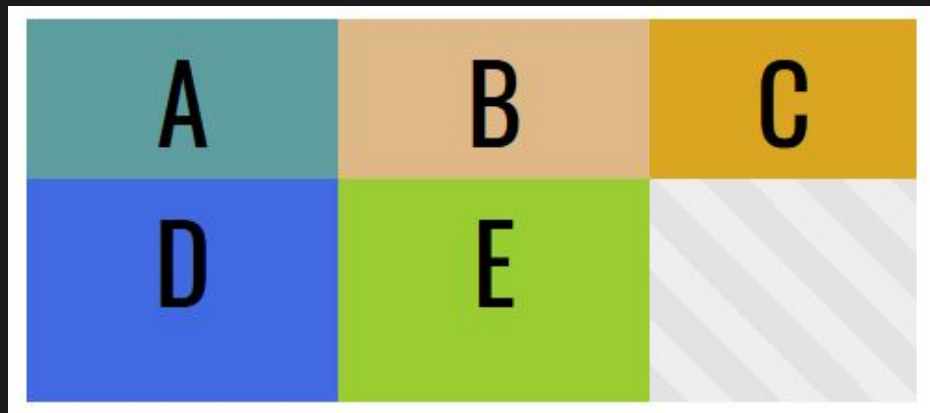


W przykładzie zdefiniowaliśmy 3 kolumny. Druga kolumna ma auto (co oznacza, tyle ile potrzebuje). Trzecia ma 30%, co oznacza 30% wielkości kontenera. 1fr, pierwsza wartość, oznacza całą pozostałą (a więc wolną) przestrzeń. Najpierw wliczone jest 30% i wartość minimalna dla auto i wartość minimalna dla fr (fr też ma minimalną wartość, którą jest zawartość, podobnie jak przy auto). To co zostaje (czyli wolna przestrzeń w kontenerze) jest przekazane do kolumn które mają wartość we frakcjach. W naszym przykładzie cała wolna przestrzeń wędruje do pierwszej kolumny (do kolumny z jednostką fr)



# Kolumny w siatce - auto

```
.grid {  
  display: grid;  
  grid-template-columns: auto auto 30%;  
  grid-template-rows: 1fr 100px;  
  //justify-content: stretch; //domyślna  
}
```



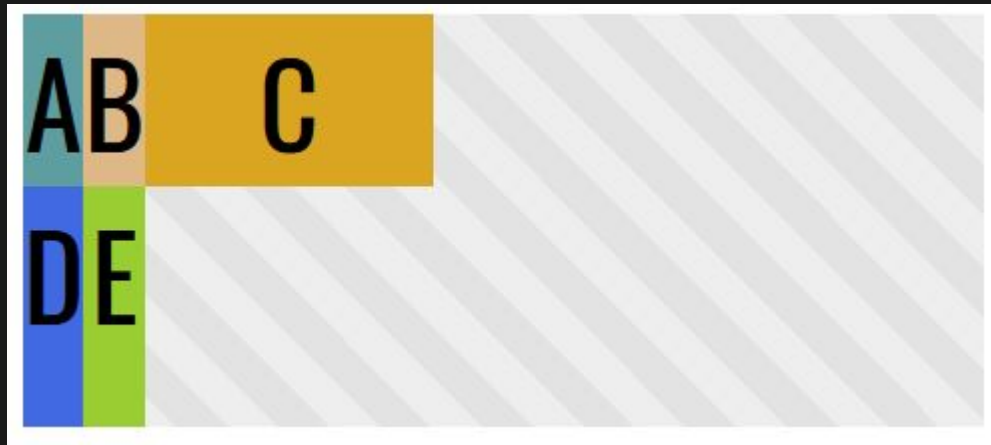
Dwie wartości auto oznaczają, że obie te kolumny będą miały taką wielkość jaką potrzebują dla swojego contentu. Ale nie do końca to jest to co widzimy, prawda!? Te dwie kolumny są wyraźnie większe niż potrzebują.

Kolumny te zostały w tym wypadku rozciągnięte ponieważ wartość `justify-content` jest ustawiona domyślnie w grid na `stretch` (rozciągnij). Elementy z wielkością `auto` będą rozciągnięte. Nie wynika to bezpośrednio z `auto`, a z tego, że element `auto` może być po prostu rozciągnięty. Element, który ma określoną, ściśle wskazaną wielkość (np. piksele, procenty itd.) już nie będzie rozciągnięty.



# bez rozciągania

```
.grid {  
  display: grid;  
  grid-template-columns: auto auto 30%;  
  grid-template-rows: 1fr 100px;  
  justify-content: left;  
}
```



Gdyby wartość `justify-content` zmienić na wartość np. `left`, to do rozciągnięcia tych kolumn (przy `auto`) by nie doszło. Teraz widzimy czym rzeczywiście jest wielkość `auto` i skąd wynika rozciąganie.



# justify-content

```
.grid {  
  display: grid;  
  grid-template-columns: auto 1fr 30%;  
  grid-template-rows: 1fr 100px;  
  justify-content: left;  
}
```



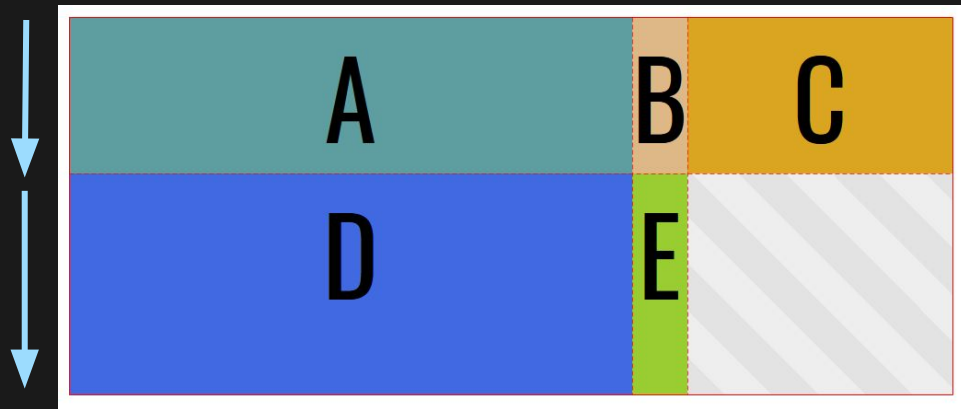
Warto jednak wiedzieć, że jeśli użyjemy zamiast auto chociaż raz jednostki fr (fraction), to justify-content nie będzie już miała zastosowania (znaczenia), ponieważ nie będzie tutaj wolnej przestrzeni. Całą wolną przestrzeń zajmie właśnie kolumna z jednostką fraction. Przydzielanie wielkości poprzedza rozciąganie za które odpowiedzialna jest właściwość justify-content.





# Wiersze w siatce

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr auto 30%;  
  grid-template-rows: 1fr 100px;  
}
```



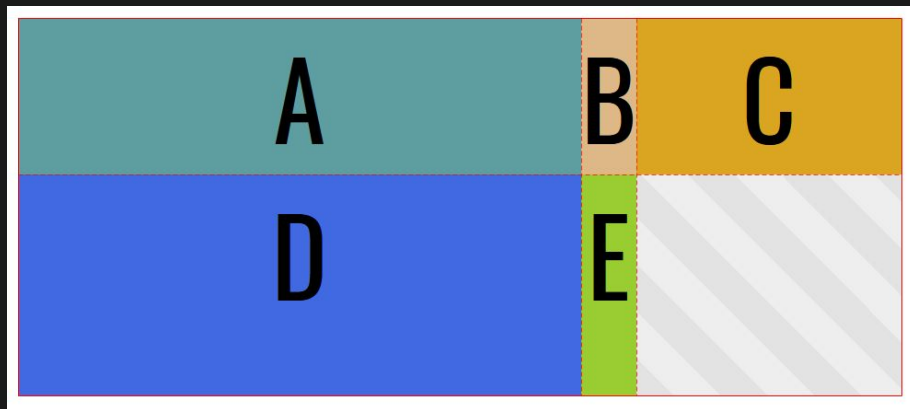
Jeśli chodzi o wiersze, to jawnie definiujemy w przykładzie dwa z których pierwszy ma jedną frakcję a drugi 100px. O ile drugi wiersz jest oczywisty, to pierwszy, określony przez jedną frakcję może budzić zastanowienie. Na szerokość kontener grid ma 100% rodzica (jeśli użyjemy `display: grid`). A ile ma na wysokość? I czy jest tu jakaś wolna przestrzeń, którą może zająć frakcja?

Na wysokość kontener ma domyślnie tyle co każdy element blokowy - więc tyle ile zajmują dzieci, w tym wypadku wiersze. Słusznie więc można powiedzieć, że żadnej wolnej przestrzeni tu nie ma. Ile zajmuje więc pierwszy wiersz wyznaczony przez 1 frakcję?



# Czym jest fraction?

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr auto 30%;  
  grid-template-rows: 1fr 100px;  
}
```

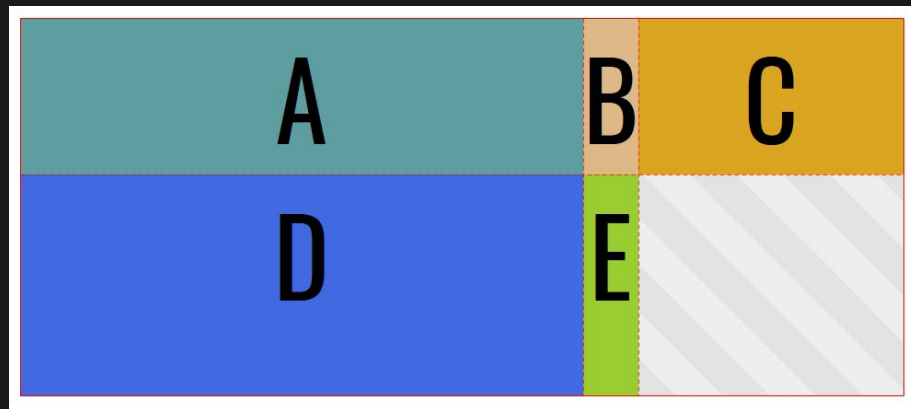


1fr zajmuje tu wielkość minimalną jaką przyjmuje frakcja czyli tyle ile potrzebuje kontent tego wiersza. Tak się więc składa, że wartość minimalna frakcji jest równa wartości auto. Wielkość frakcji to zawsze wielkość minimalna czyli wielkość której potrzebuje zawartość, plus, ewentualnie, dostępna wolna przestrzeń. Ponieważ w naszym przykładzie nie ma wolnej przestrzeni to 1fr oznacza w tym wypadku dokładnie to co auto.



# 1fr = auto ?

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr auto 30%;  
  grid-template-rows: auto 100px;  
}
```



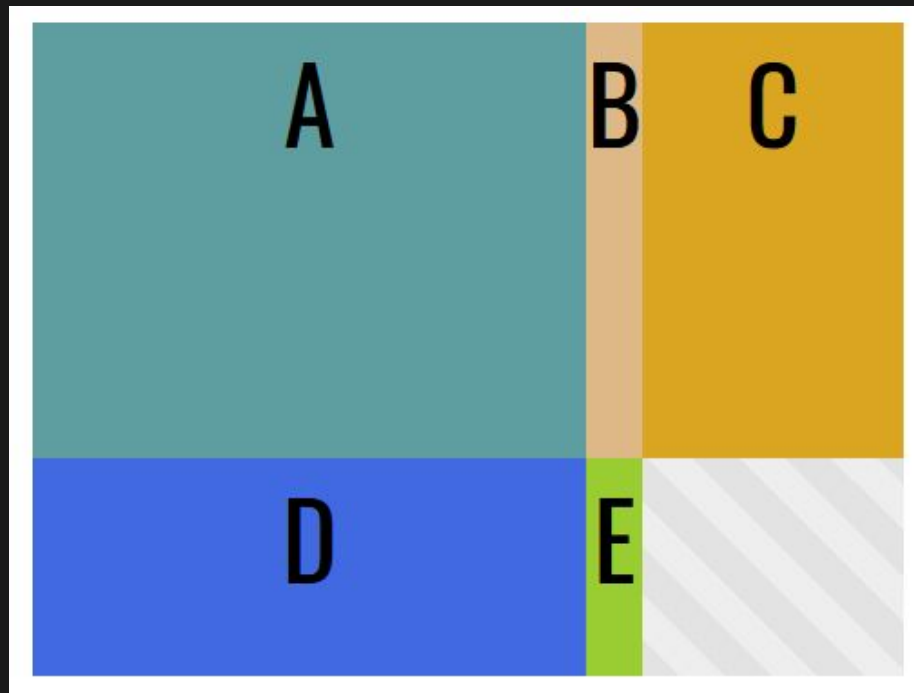
Zamiana 1fr na auto nic nie zmienia jeśli nie ma wolnej przestrzeni.



# Wolna przestrzeń

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr auto 30%;  
  grid-template-rows: 1fr 100px;  
  height: 300px;  
}
```

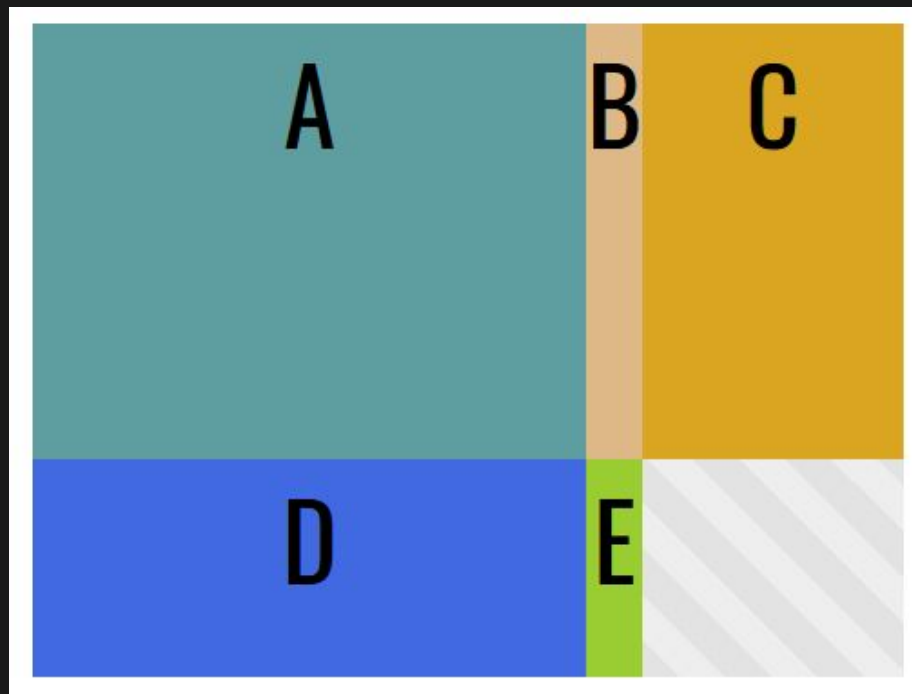
Siłą frakcji w pionie (wielkość wiersza) zobaczymy gdy zwiększymy (stworzymy) wolną przestrzeń np. ustawiając wysokość kontenera. 1fr zajmie całą przestrzeń, która w ten sposób powstała, o ile oczywiście jest jedyną taką wartością w wielkościach wierszy. W praktyce pierwszy wiersz ma teraz 200px wysokości.



# rozciąganie auto

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr auto 30%;  
  grid-template-rows: auto 100px;  
  height: 300px;  
  // align-content: stretch;  
}
```

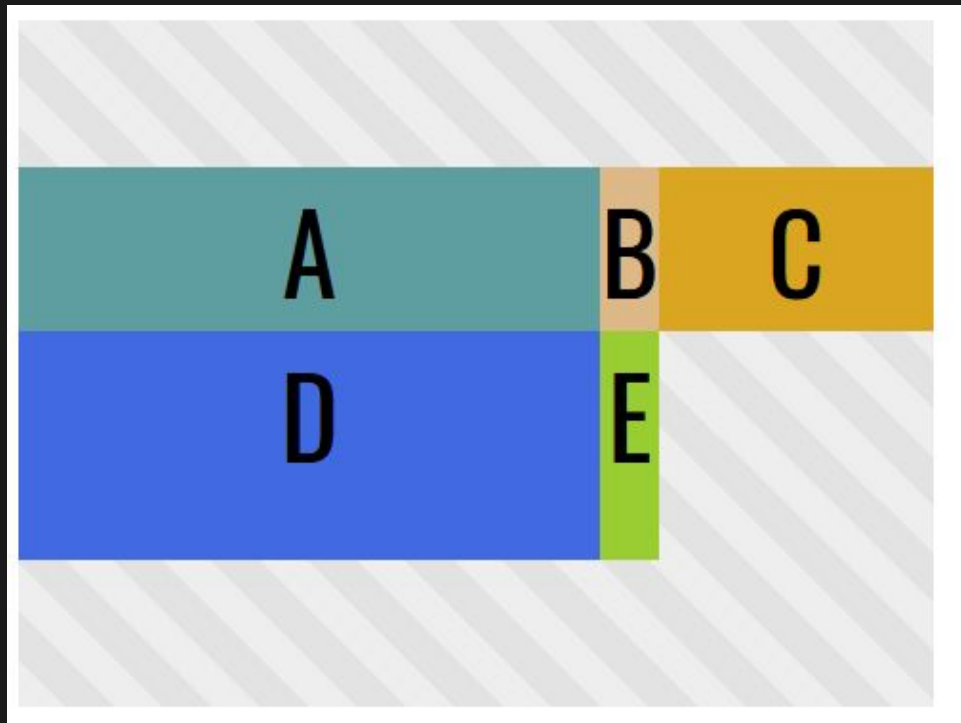
Jeśli w tym konkretnym przykładzie zamienimy 1fr na auto, to efekt będzie taki sam, ponieważ auto jest dodatkowo rozciągane przez właściwość align-content, która domyślnie jest ustawiona na align-content: stretch.



# align-content

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr auto 30%;  
  grid-template-rows: auto 100px;  
  height: 300px;  
  align-content: center;  
}
```

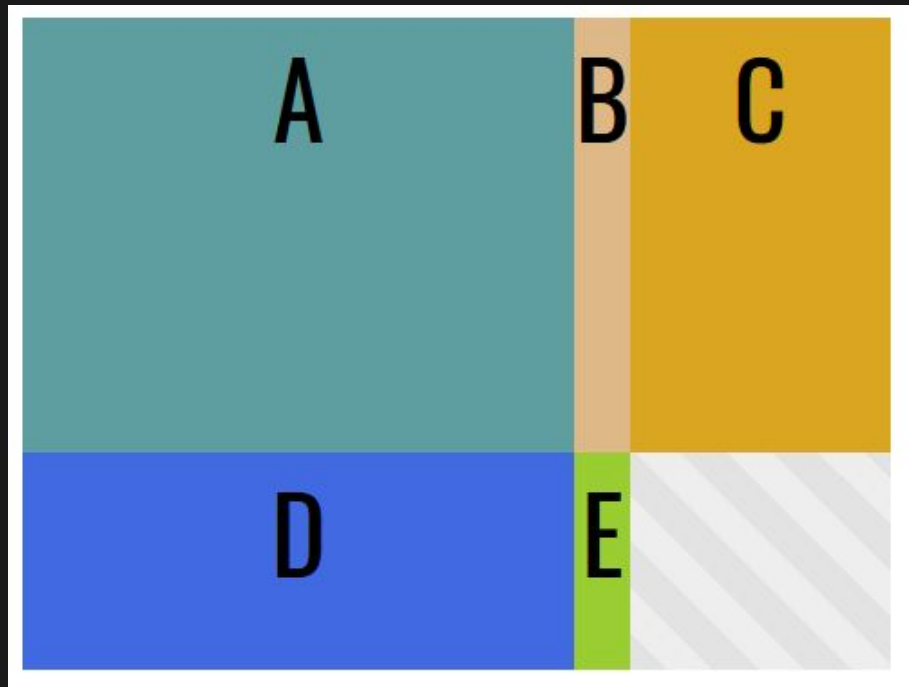
Jeśli zmienimy domyślne align-content ze stretch na np. center, to zobaczymy, że auto wcale nie rozciąga wiersza (w przeciwieństwie do fraction).



# fraction ma pierwszeństwo

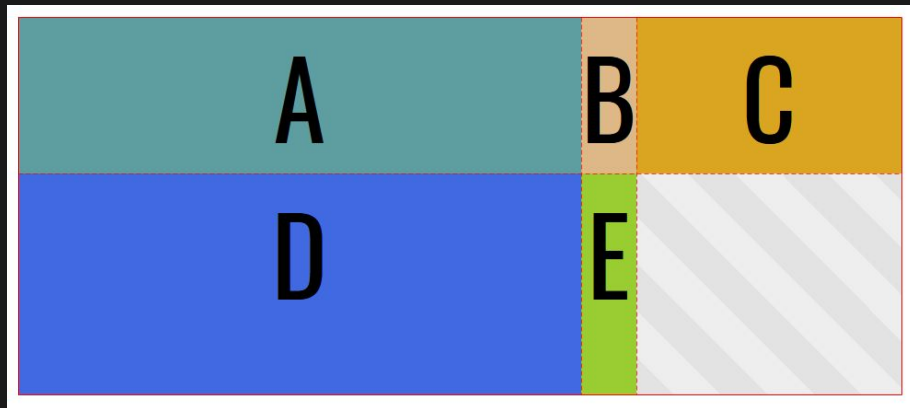
```
.grid {  
  display: grid;  
  grid-template-columns: 1fr auto 30%;  
  grid-template-rows: 1fr 100px;  
  height: 300px;  
  align-content: center;  
}
```

Jeśli przywrócimy z auto na 1fr to zobaczymy, że w takim wypadku align-content już nie ma żadnego wpływu na układ. Align-content też dystrybuuje wolną przestrzeń, ale fraction robi to w pierwszej kolejności. Jeśli zależy nam by coś się rozciągało na wolną przestrzeń to użyjmy fraction zamiast polegać na auto.



# Wartości w siatce

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr auto 30%;  
  grid-template-rows: 1fr 100px;  
}
```



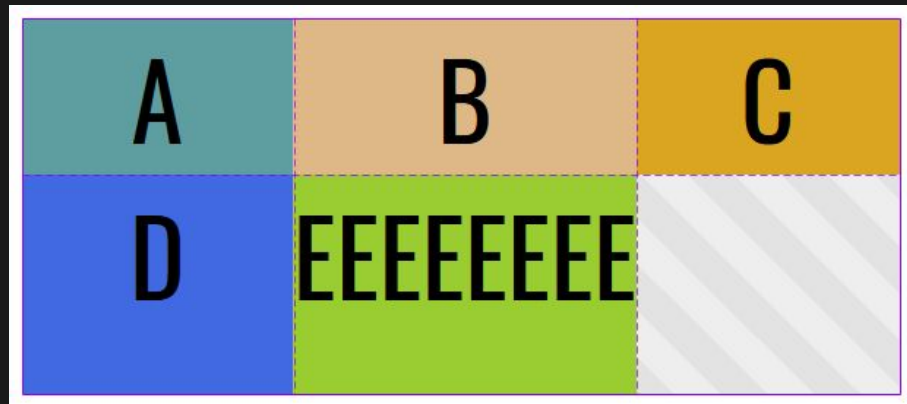
Wróćmy do naszego podstawowego ustawienia i sprawdźmy jeszcze kilka zachowań.





# auto dostosowuje się do rozmiaru

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr auto 30%;  
  grid-template-rows: 1fr 100px;  
}
```

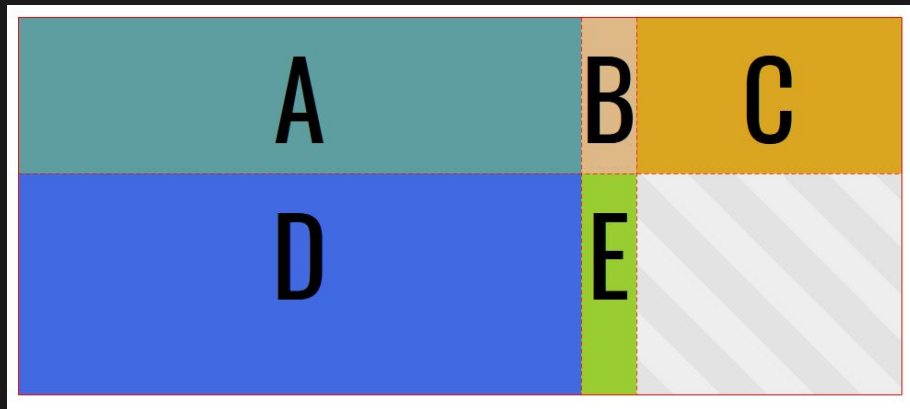


W przykładzie widzimy jak zareaguje układ gdy zwiększymy zawartość kolumny drugiej. Auto oznacza, że kolumna dopasuje się do największego elementu w tej kolumnie. Zmniejszeniu ulegnie przy tym wolna przestrzeń, czyli w naszym przykładzie zmniejszy się kolumna numer 1.



# Wartości w siatce

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr auto 30%;  
  grid-template-rows: 1fr 100px;  
}
```

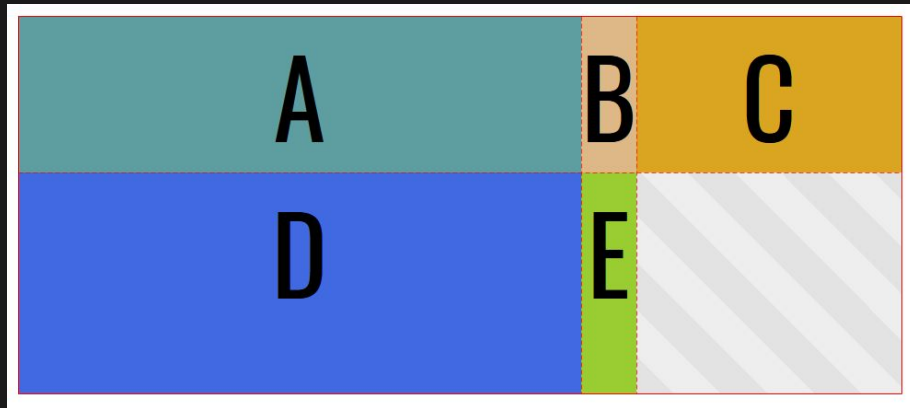


Wróćmy do podstawowego ustawienia i zobaczmy na kolejnym slajdzie co się stanie gdy zwiększymy wartość dla jednostki fraction.



# fr w kolumnach i wierszach - inne

```
.grid {  
  display: grid;  
  grid-template-columns: 2fr auto 30%;  
  grid-template-rows: 50fr 100px;  
}
```

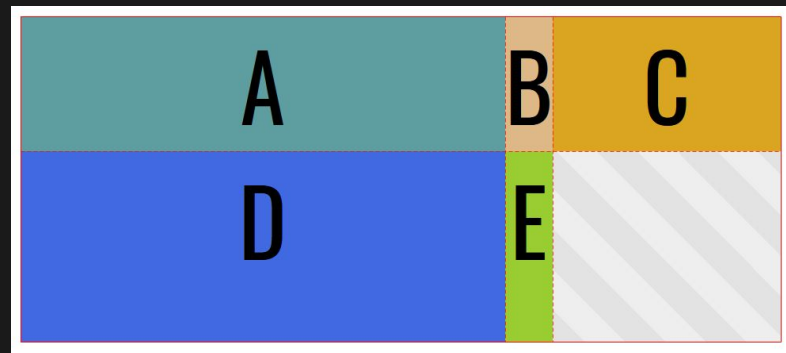


Po pierwsze pamiętajmy, że oczywiście jednostki fraction w kolumnach i wierszach odwołują się do innej przestrzeni. Wolna przestrzeń jest liczona osobno dla kolumn i wierszy.



# 50fr 2fr?

```
.grid {  
  display: grid;  
  grid-template-columns: 2fr auto 30%;  
  grid-template-rows: 50fr 100px;  
}
```



Jak działa taki zapis (jak wyżej)? Jak najbardziej poprawnie, choć nic nie wnosi w tym przypadku.

Wolna przestrzeń zostanie (wirtualnie) podzielona na tyle jednostek ile zdefiniujemy. W naszym przykładzie wolna przestrzeń w poziomie zostanie podzielona na 2 części (frakcje) a w pionie na 50 części. Jednak wszystkie te części przypiszemy ciągle tylko do jednej kolumny (i w drugim wymiarze do wierszy), więc nie spowoduje to żadnej zmiany. Nawet możemy tam wpisać 1000fr i tak nic to nie zmieni. To tak jak podzielić ciasteczko na 5 kawałków i 1000 kawałków. Od tego ciasto nie stanie się większe/mniejsze.

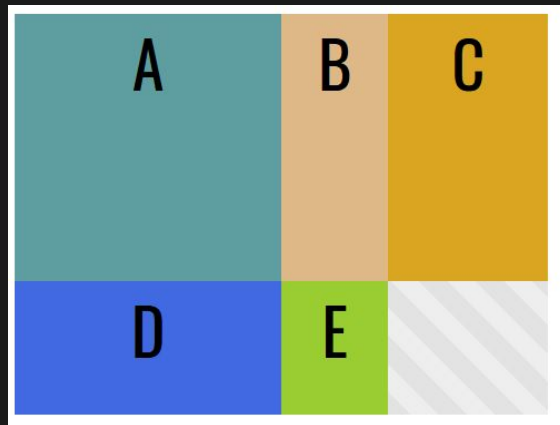
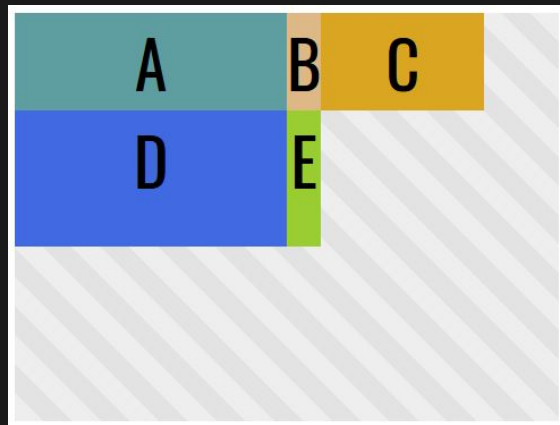


# Wolna przestrzeń i jej podział

```
.grid {  
  display: grid;  
  grid-template-columns: 200px 1000fr 30%;  
  grid-template-rows: 3fr 100px;  
  height: 300px;  
}
```

Przestrzeń po prawej od C to wolna przestrzeń, która może być rozdzielona między frakcje w kolumnach.

Przestrzeń poniżej wiersza w którym jest D i E to wolna przestrzeń, która może być rozdzielona między frakcje w wierszach.



Przed przydzieleniem wolnej przestrzeni

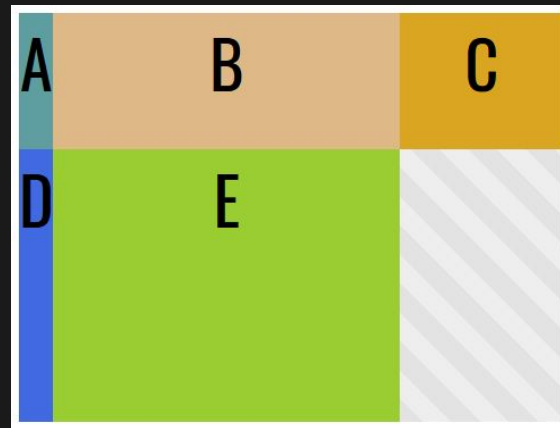
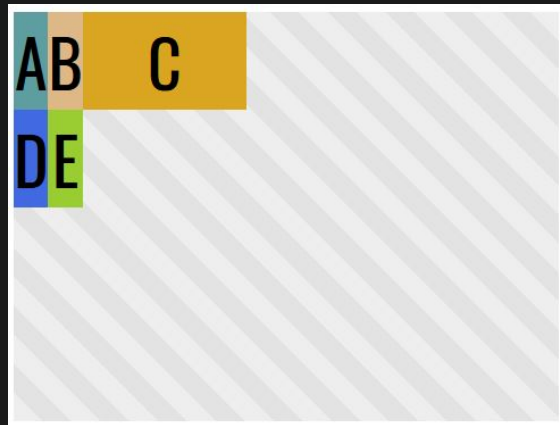
Po przydzieleniu wolnej przestrzeni



Więcej niż jedna wartość fr w danym wymiarze - kolumny

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr 999fr 30%;  
  grid-template-rows: 5fr 10fr;  
  height: 300px;  
}
```

Wolna przestrzeń w poziomie podzielona na 1001 kawałków. Pierwsza kolumna ma wielkość kontentu + 1/1000 wolnej przestrzeni a druga kolumna wielkość minimalna i 999/1000 wolnej przestrzeni.



Przed przydzieleniem wolnej przestrzeni

Po przydzieleniu wolnej przestrzeni

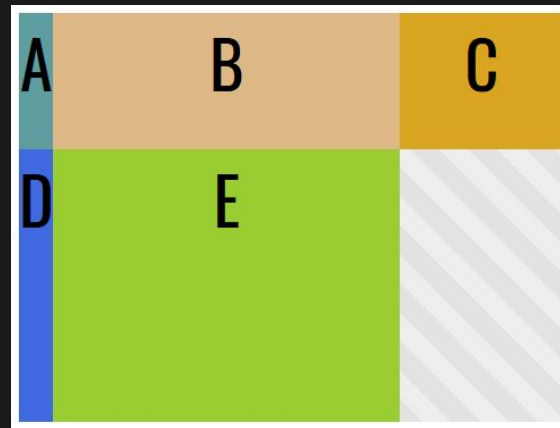
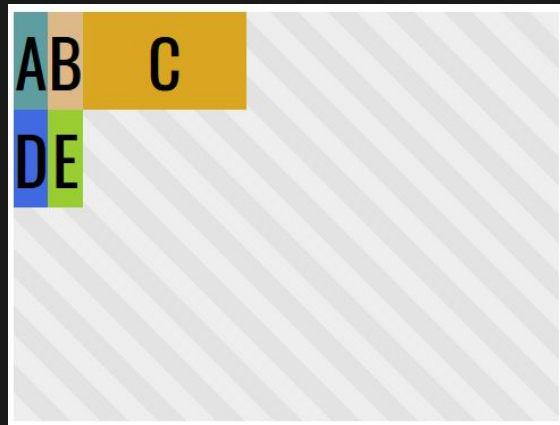


WEBSAMURAJ

Więcej niż jedna wartość fr w danym wymiarze - wiersze

```
.grid {  
  display: grid;  
  grid-template-columns: 1fr 100fr 30%;  
  grid-template-rows: 5fr 10fr;  
  height: 300px;  
}
```

Wolna przestrzeń w pionie jest wirtualnie podzielona na 15 kawałków. Pierwszy wiersz ma wielkość kontentu +  $\frac{1}{3}$  (5/15) wolnej przestrzeni a drugi wiersz wielkość minimalną (wielkość kontentu) i  $\frac{2}{3}$  wolnej przestrzeni.



Przed przydzieleniem wolnej przestrzeni

Po przydzieleniu wolnej przestrzeni

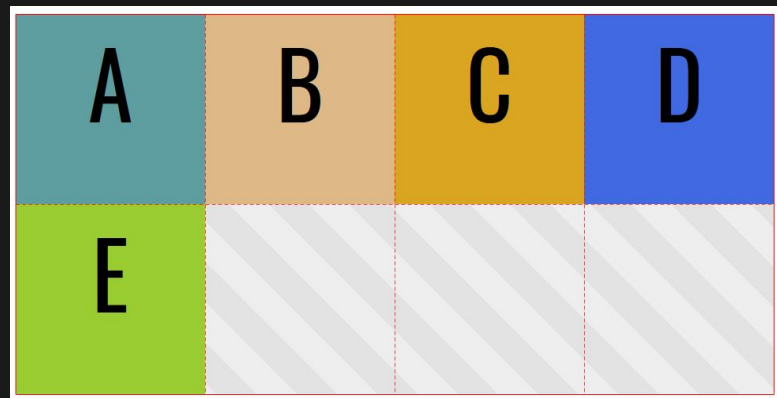


WEBSAMURAJ

# Procenty jako wartość w kolumnach

```
.grid {  
  display: grid;  
  grid-template-columns: 25% 25% 25% 25%;  
  grid-template-rows: 100px 100px;  
}
```

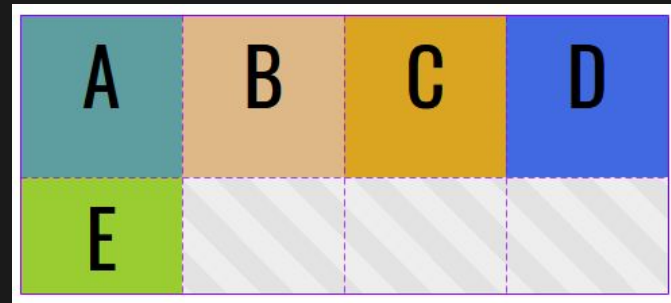
Procent odnosi się do wielkości kontenera. W przypadku kolumn sytuacja jest prosta. Jeśli kontener ma 600px to każda z kolumn w tym przykładzie będzie miała 150px.





# Procenty jako wartość wierszy

```
.grid {  
  display: grid;  
  grid-template-columns: 25% 25% 25% 25%;  
  grid-template-rows: 100px 1500%;  
}
```



W przypadku wysokości sytuacja nie jest już taka prosta (dokładnie tak samo jak to mam miejsce bez grida). Przeglądarki zachowują się tu różnie. W Firefoxie do wysokości nie ma sensu odnosić się procentowo, jeśli ta wysokość nie jest zadeklarowana, ponieważ domyślna wysokość wynosi 0 (oczywiście jest większa, ale to wynika z wielkości elementów w kontenerze). To czy napiszemy 15% czy 1500% nie ma znaczenia w takim przypadku. Wiersz będzie miał wielkość uzależnioną od zawartości (minimalną potrzebną). W Chrome ta wartość zostanie jednak przeliczona i np. 1500% będzie odnosić się do wysokości kontenera wyliczonej wg wartości (w naszym przykładzie) 100px i auto (tyle ile potrzebuje). W praktyce 1500% razy gdzieś 170px (suma dwóch wierszy) czyli gdzieś 2550px.

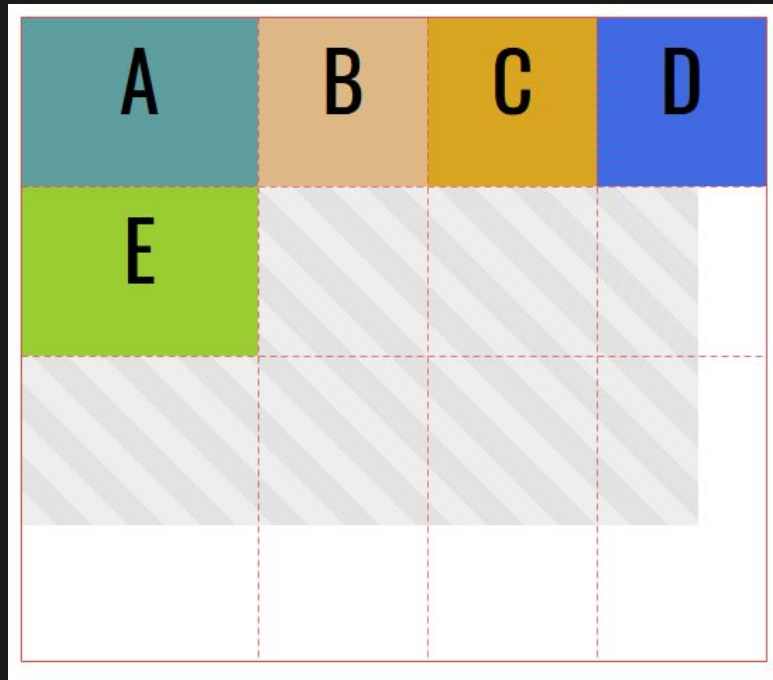


# Deklaracja wysokości

```
.grid {  
  display: grid;  
  grid-template-columns: 35% 25% 25% 25%;  
  grid-template-rows: 100px 100px 60%;  
  height: 300px;  
}
```

Sytuacji zmieni się gdy zadeklarujemy wysokość kontenera.

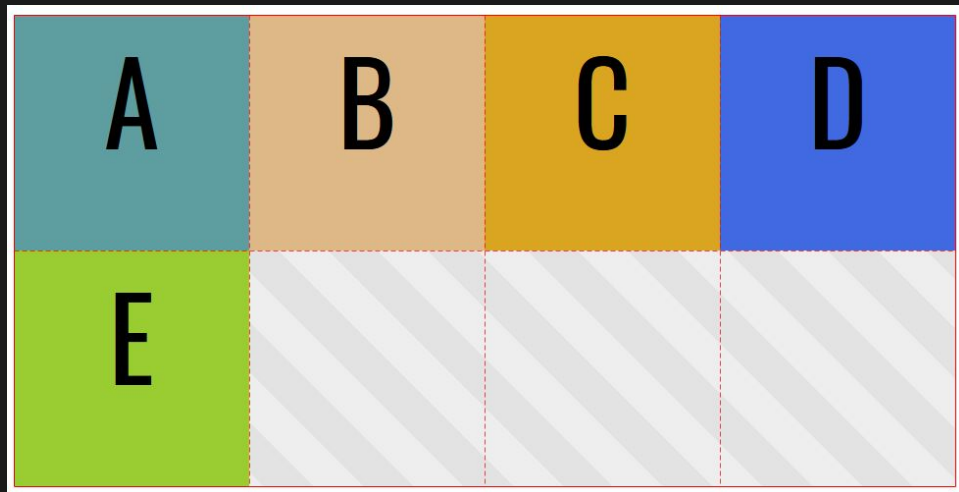
Przy okazji zobaczmy co stanie się z siatką gdy suma procentów (w kolumnach) i suma wielkości (procenty + px) będzie większa niż kontener siatki.



# Frakcje a procenty

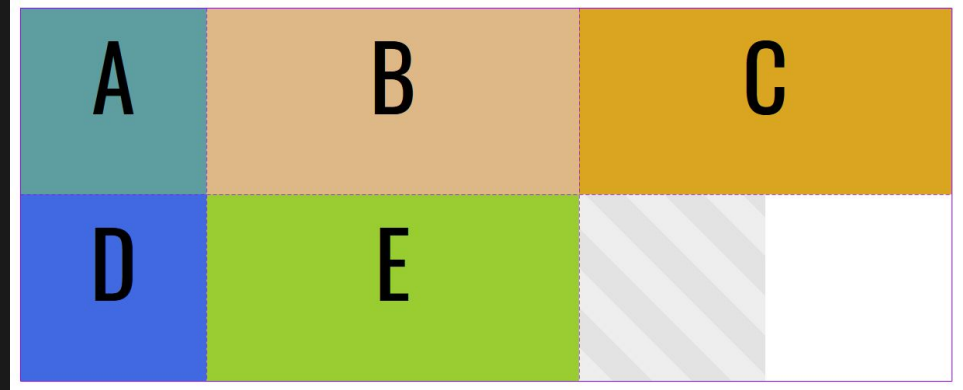
```
.grid {  
  display: grid;  
  grid-template-columns: 1fr 1fr 1fr 1fr;  
  grid-template-rows: 100px 100px;  
}
```

W tym wypadku efekt będzie taki sam jak użycie procentów. Ale używanie fraction jest zazwyczaj lepszym pomysłem.



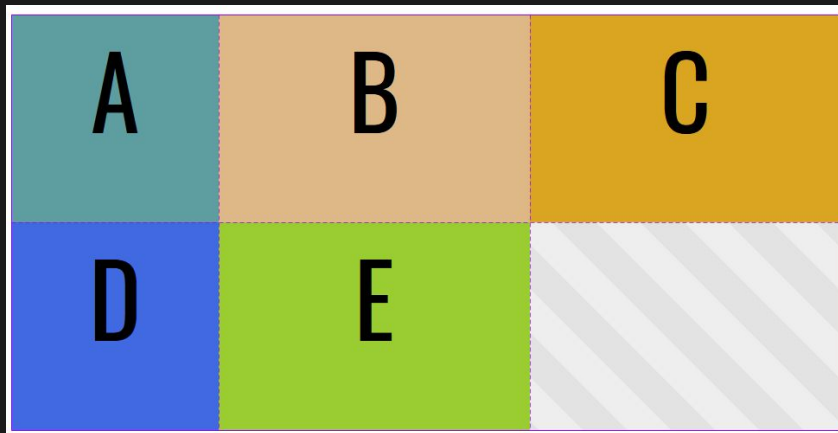
# Problem z procentem

```
.grid {  
  display: grid;  
  grid-template-columns: 100px 50% 50%;  
  grid-template-rows: 100px 100px;  
}
```



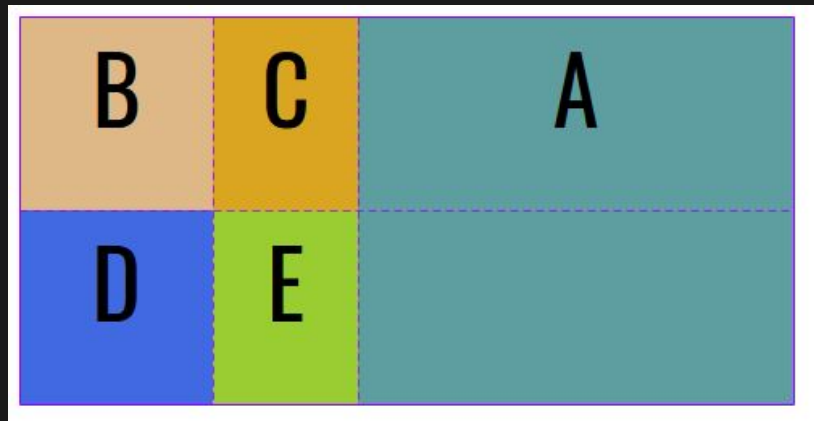
# Rozwiązanie z jednostkami frakcji

```
.grid {  
  display: grid;  
  grid-template-columns: 100px 1fr 1fr;  
  grid-template-rows: 100px 100px;  
}
```



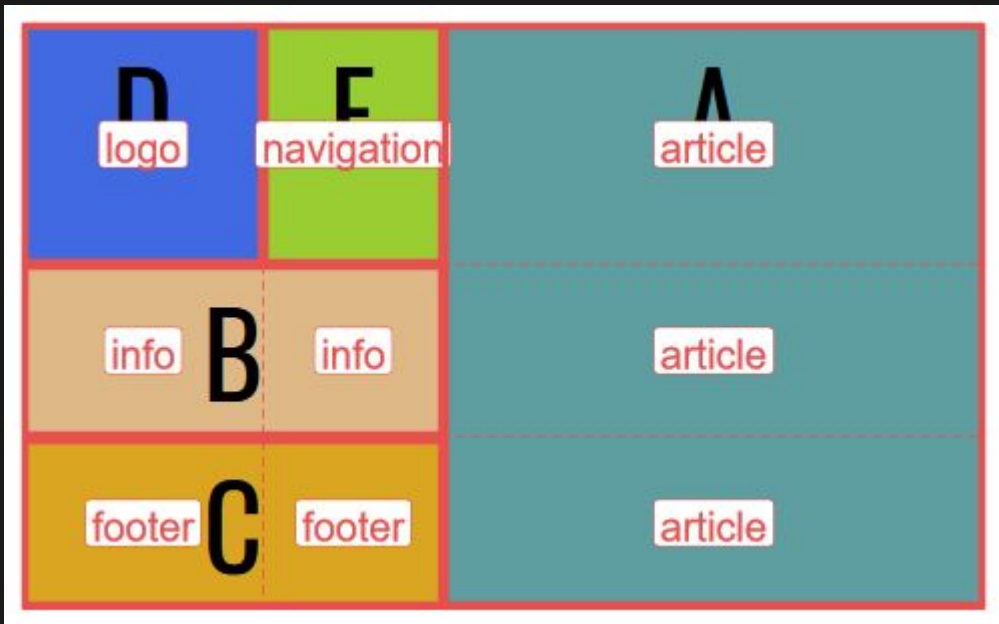
# Definiowanie siatki i pozycjonowanie elementów w siatce

```
.grid {  
  display: grid;  
  grid-template-columns: 100px 1fr 3fr;  
  grid-template-rows: 100px 100px;  
}  
/* A */  
.e1 {  
  grid-area: 1/3/3/4;  
  //grid-row-start: 1;  
  //grid-column-start: 3;  
  //grid-row-end: 3;  
  //grid-column-end: 4;  
}
```



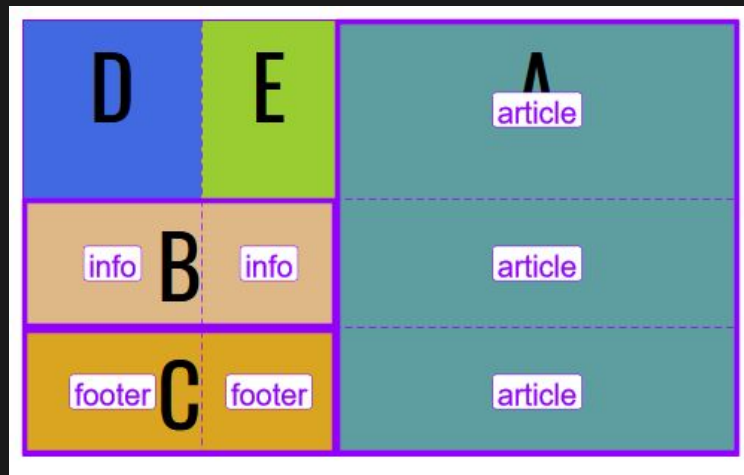
# grid-template-areas - nazywanie komórek

```
.grid {  
  display: grid;  
  grid-template-columns: 100px 1fr 3fr;  
  grid-template-rows: 100px auto auto;  
  grid-template-areas:  
    "logo navigation article"  
    "info info article"  
    "footer footer article";  
}  
  
.e1 { grid-area: article; }  
.e2 { grid-area: info; }  
.e3 { grid-area: footer; }
```



# grid-template-areas - nazywanie komórek

```
.grid {  
  display: grid;  
  grid-template-columns: 100px 1fr 3fr;  
  grid-template-rows: 100px 100px;  
  grid-template-areas:  
    " . . article"  
    "info info article"  
    "footer footer article";  
}  
  
.e1 { grid-area: article; }  
.e2 { grid-area: info; }  
.e3 { grid-area: footer; }
```



ps. pisząc grid-area w takiej formie tak naprawdę piszemy 4 wartości.

grid-area: info/info/info/info;

za każdą z tych pozycji stoi grid-row-start / grid-column-start / grid-row-end / grid-column-end;







# Zabawa z przykładami

- właściwości kontenera CSS Grid



# Właściwości kontenera

`grid-template-columns` // liczba kolumn i ich wielkość

`grid-template-rows` // liczba wierszy i ich wielkość

`grid-template-areas` // nazwy komórek

`grid-auto-columns` // wielkość automatycznie tworzonych kolumn (auto)

`grid-auto-rows` // wielkość automatycznie tworzonych wierszy (auto)

`grid-auto-flow` // w jaki sposób automatycznie będą dodawane elementy siatki (i jak będzie tworzona automatyczna siatka) - domyślnie row (można też użyć column). Można użyć oprócz nich wartości dense.

`justify-content` // jak będą zachowywać się kolumny gdy będzie wolna przestrzeń w kontenerze (szerokość)  
np. left, right, center, start, end, space-between, space-around, space-evenly i stretch

`align-content` // jak będą zachowywać się wiersze gdy będzie wolna przestrzeń w kontenerze (wysokość).  
Wartości jak w justify-content, ale bez left i right;

`place-content` // skrót - możemy napisać jednocześnie wartości dla align-content i justify-content. Najpierw podajemy align-content a potem justify-content tj. `place-content: align-content justify-content;`



# Właściwości kontenera

column-gap (grid-column-gap)

row-gap (grid-row-gap)

gap (grid-gap)

grid-template

grid

justify-items

align-items

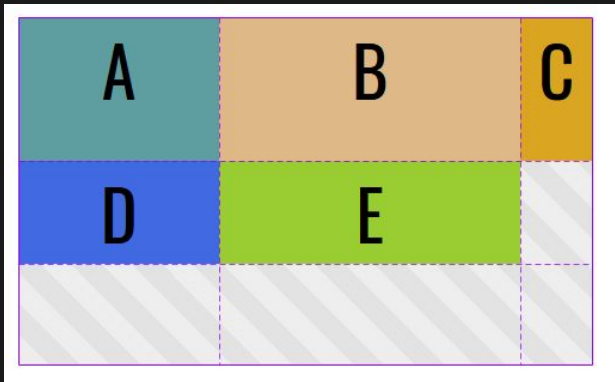
place-items



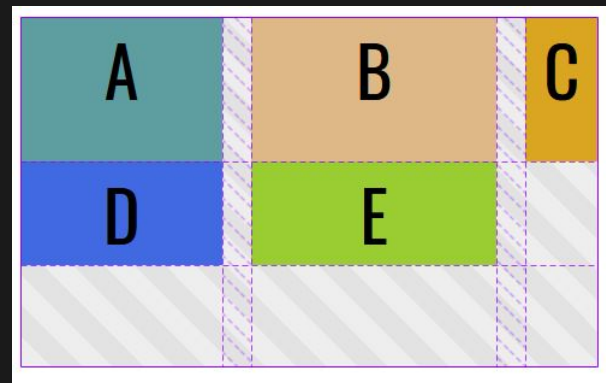
## column-gap (grid-column-gap)

```
.grid {  
  display: grid;  
  grid-template-columns: 140px 1fr 50px;  
  grid-template-rows: 100px auto 70px;  
  column-gap: 20px;  
}
```

Przed dodanie column-gap



Po dodaniu column-gap

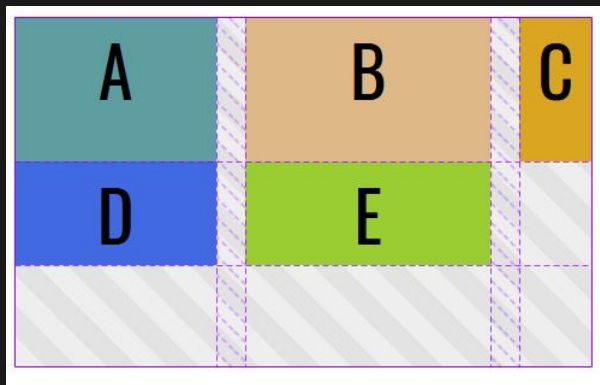


## column-gap (grid-column-gap)

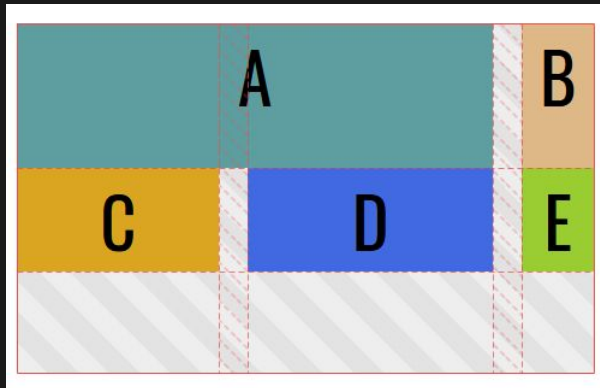
```
.grid {  
  display: grid;  
  grid-template-columns: 140px 1fr 50px;  
  grid-template-rows: 100px auto 70px;  
  column-gap: 20px;  
}
```

```
.e1 {  
  grid-column: 1/3;  
}
```

Przed określeniem pozycji elementy



Po określeniu pozycji elementu

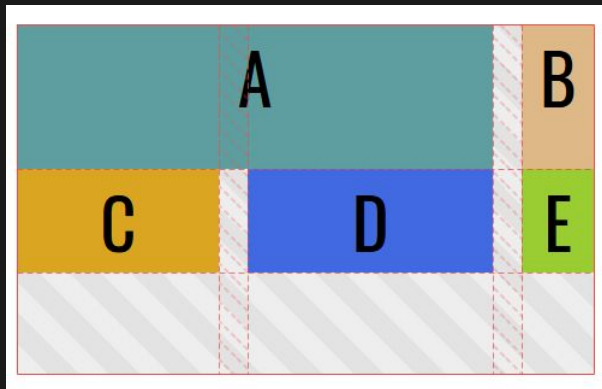


## row-gap (grid-column-gap)

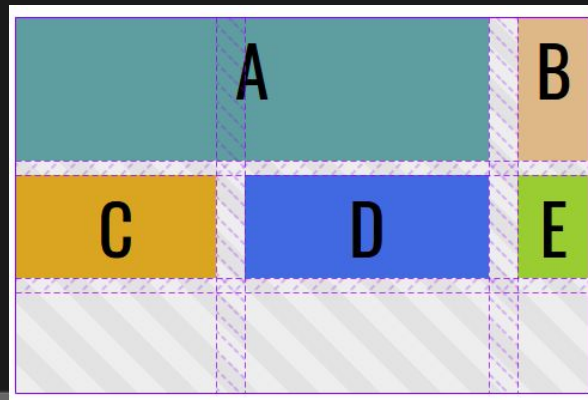
```
.grid {  
  display: grid;  
  grid-template-columns: 140px 1fr 50px;  
  grid-template-rows: 100px auto 70px;  
  column-gap: 20px;  
  row-gap: 20px;  
}
```

```
.e1 {  
  grid-column: 1/3;  
}
```

Przed określeniem row-gap



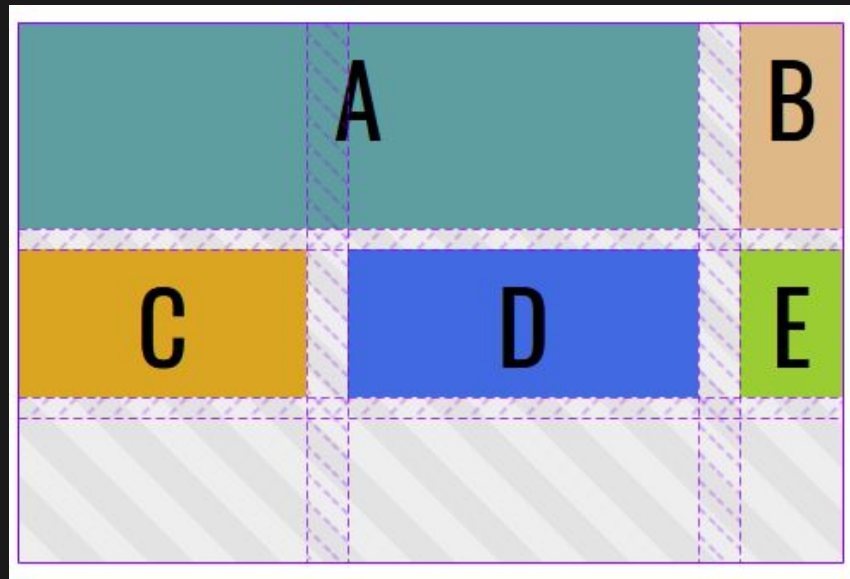
Po określeniu row-gap



## gap (grid-gap)

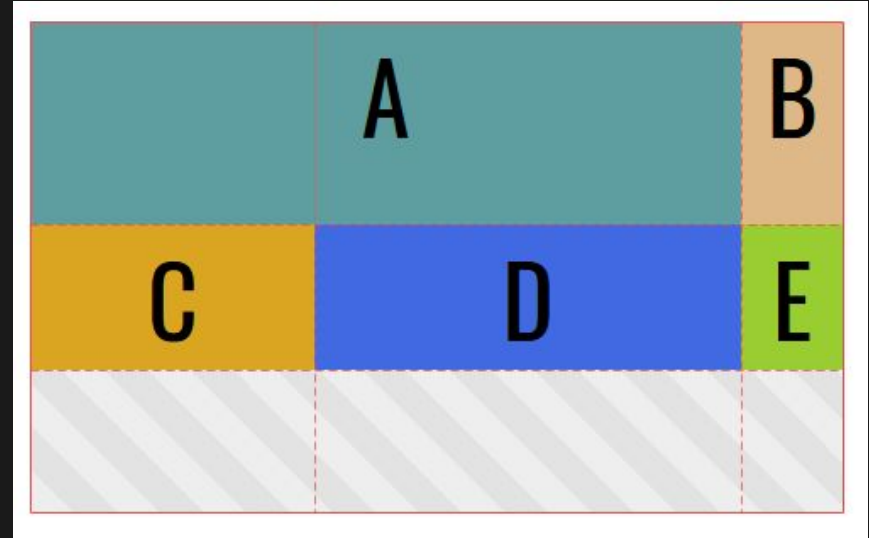
```
.grid {  
  display: grid;  
  grid-template-columns: 140px 1fr 50px;  
  grid-template-rows: 100px auto 70px;  
  // column-gap: 20px;  
  // row-gap: 10px;  
  gap: 10px 20px; // jeśli jedna wartość  
  to ta sama wtedy dla row-gap i column-gap  
}
```

```
.e1 {  
  grid-column: 1/3;  
}
```



## grid-template - grid-template-rows / grid-template-columns

```
.grid {  
  display: grid;  
  // grid-template-columns: 140px 1fr 50px;  
  // grid-template-rows: 100px auto 70px;  
  grid-template: 100px auto 70px / 140px 1fr 50px;  
}  
  
.e1 {  
  grid-column: 1/3;  
}
```

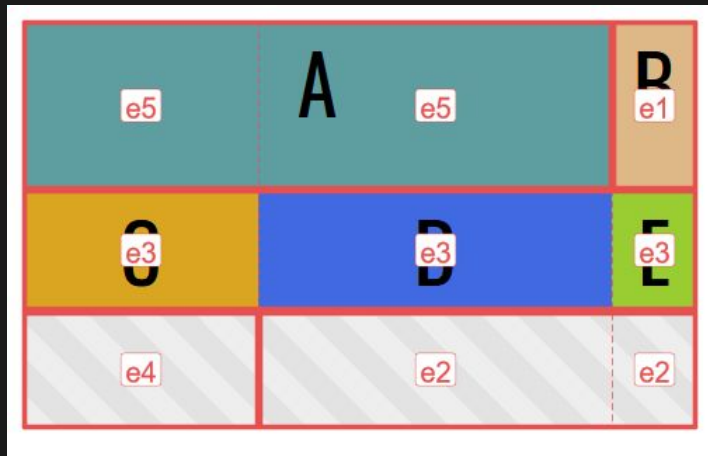




## grid - łączy grid-areas, grid-template-rows i grid-template-columns

```
.grid {  
  display: grid;  
  // grid-template: 100px auto 70px / 140px 1fr 50px;  
  grid: "e5 e5 e1" 100px "e3 e3 e3" auto "e4 e2 e2" 70px / 140px 1fr 50px;  
}
```

```
.e1 {  
  grid-column: 1/3;  
}
```

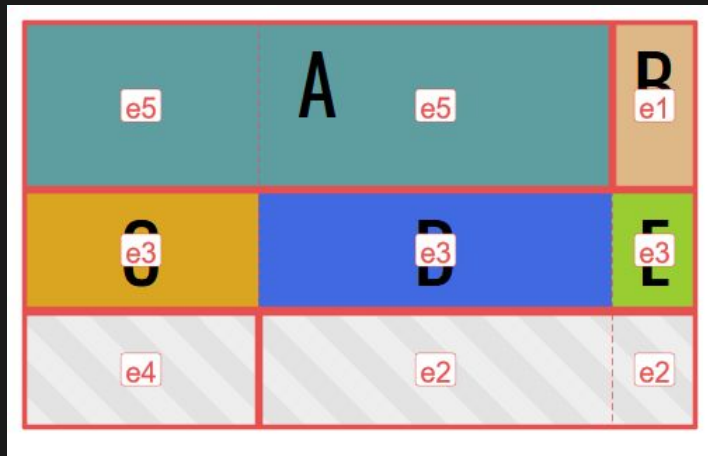


## grid - łączy grid-areas, grid-template-rows i grid-template-columns

```
.grid {  
  display: grid;  
  // grid-template: 100px auto 70px / 140px 1fr 50px;  
  grid: "e5 e5 e1" 100px "e3 e3 e3" "e4 e2 e2" 70px / 140px 1fr 50px;  
}
```

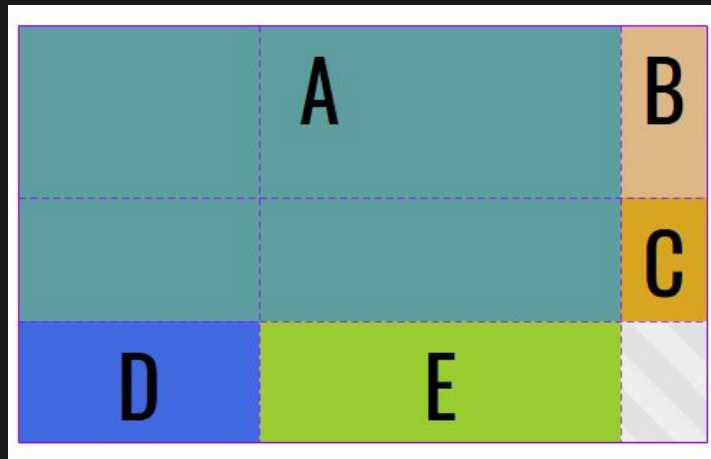
```
.e1 {  
  grid-column: 1/3;  
}
```

//wartości auto nie trzeba podawać



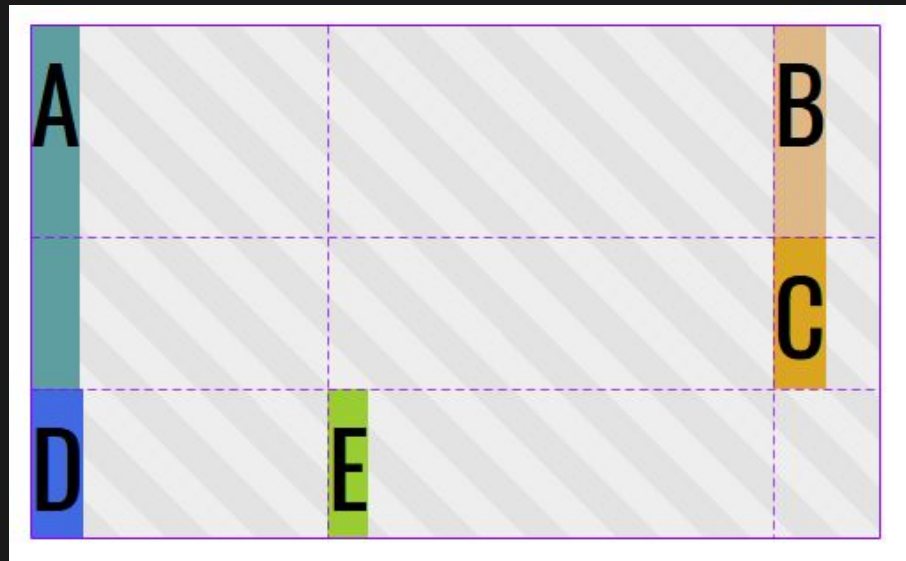
## justify-items i align-items - zachowanie wewnątrz komórki w poziomie

```
.grid {  
  display: grid;  
  grid-template-rows: 100px auto 70px;  
  grid-template-columns: 140px 1fr 50px;  
  // justify-items: stretch;  
  // align-items: stretch;  
}  
  
.e1 {  
  grid-area: 1/1/3/3;  
}
```



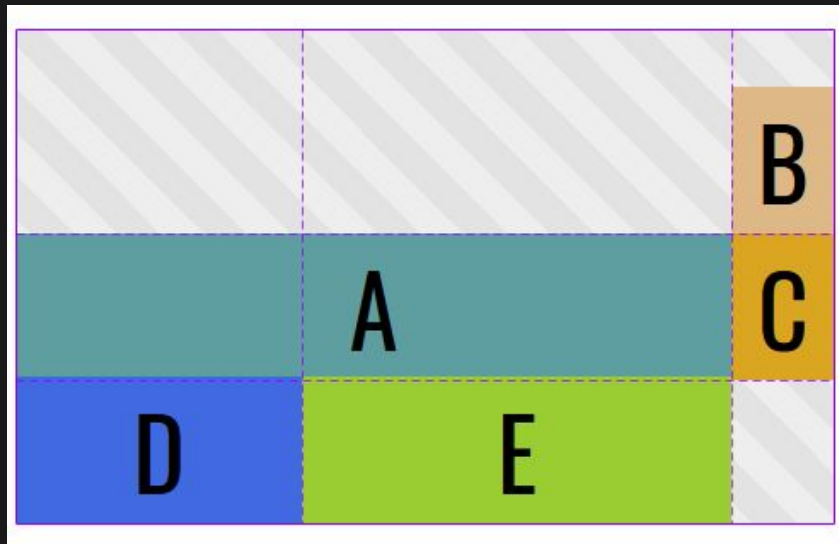
## justify-items - zachowanie elementu wewnątrz komórki w poziomie

```
.grid {  
  display: grid;  
  grid-template-rows: 100px auto 70px;  
  grid-template-columns: 140px 1fr 50px;  
  // align-items: stretch;  
  justify-items: left;  
  //start, center, end, left, right, stretch  
}  
  
.e1 {  
  grid-area:1/1/3/3;  
}
```



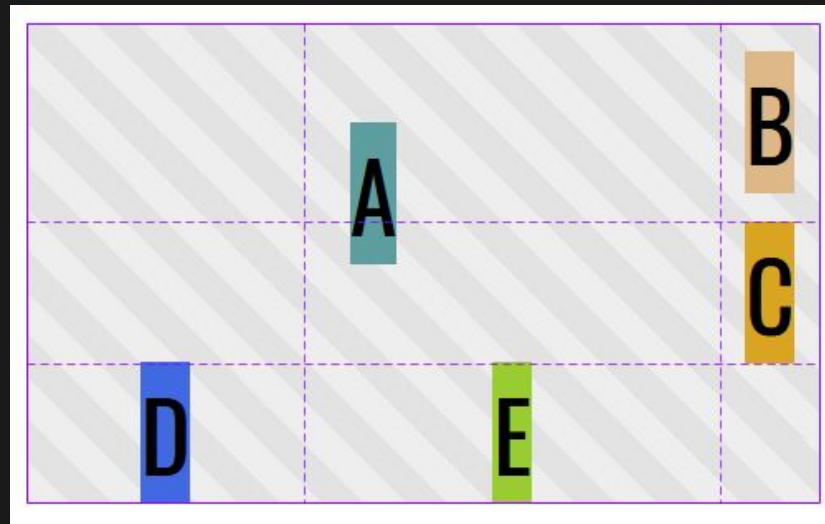
## align-items - zachowanie elementu wewnątrz komórki w pionie

```
.grid {  
  display: grid;  
  grid-template-rows: 100px auto 70px;  
  grid-template-columns: 140px 1fr 50px;  
  // justify-items: center;  
  align-items: end;  
  //start, center, end, stretch  
}  
  
.e1 {  
  grid-area:1/1/3/3;  
}
```



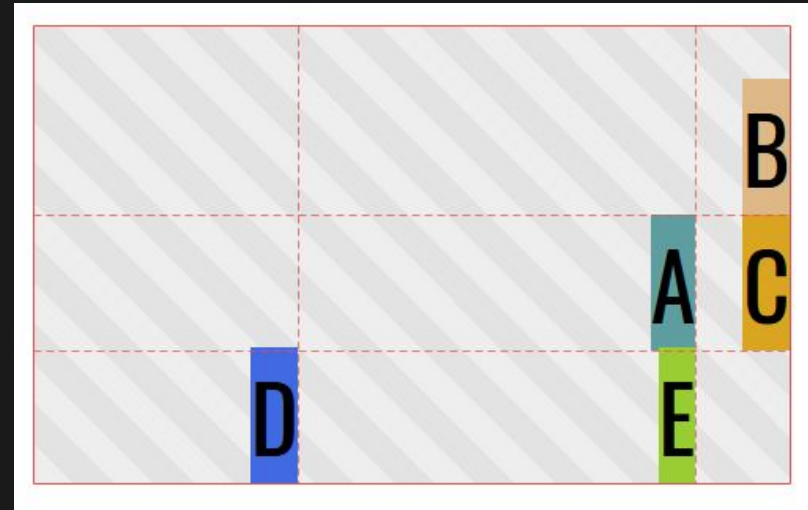
## align-items i justify-items razem

```
.grid {  
  display: grid;  
  grid-template-rows: 100px auto 70px;  
  grid-template-columns: 140px 1fr 50px;  
  justify-items: center;  
  align-items: center;  
}  
  
.e1 {  
  grid-area: 1/1/3/3;  
}
```



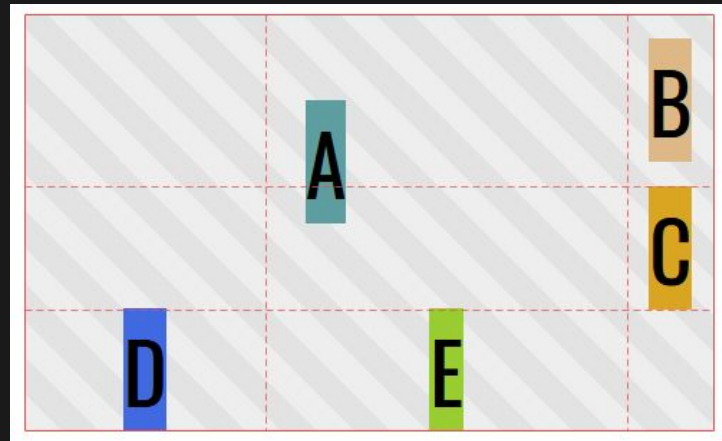
## place-items - skrót dla align-items i justify-items

```
.grid {  
  display: grid;  
  grid-template-rows: 100px auto 70px;  
  grid-template-columns: 140px 1fr 50px;  
  place-items: end right;  
  //place-items: align-items justify-items;  
}  
  
.e1 {  
  grid-area: 1/1/3/3;  
}
```



## place-items - skrót dla align-items i justify-items

```
.grid {  
  display: grid;  
  grid-template-rows: 100px auto 70px;  
  grid-template-columns: 140px 1fr 50px;  
  place-items: center;  
}  
  
.e1 {  
  grid-area: 1/1/3/3;  
}
```







# Zabawa z przykładami

- właściwości elementu siatki



# Właściwości elementu, które poznaliśmy

`grid-column-start` // od której linii pionowej zaczyna się powierzchnia danego elementu (definiujemy kolumnę)

`grid-column-end` // na której linii pionowej kończy się powierzchnia danego elementu (definiujemy kolumnę)

`grid-row-start` // na której linii poziome zaczyna się powierzchnia danego elementu (definiujemy wiersz)

`grid-row-end` // na której linii poziome kończy się powierzchnia danego elementu (definiujemy wiersz)

`grid-column` // wersja skrócona dla `grid-column-start` i `grid-column-end` np. `grid-column: 2/6`

`grid-row` // wersja skrócona dla `grid-row-start` i `grid-row-end` np. `grid-row: 2/6`

`grid-area` // wersja skrócona dla wszystkich czterech właściwości.

`grid-area: grid-row-start / grid-column-start / grid-row-end / grid-column-end` np. `grid-area: 2/2/span 3/5;`

`grid-area: navigation`



# Właściwości elementu, które jeszcze musimy poznać

align-self

justify-self

place-self

order

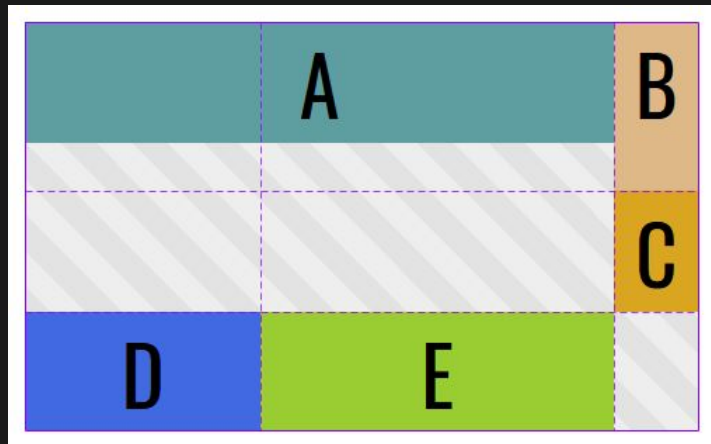
z-index



## align-self i justify-self - dla poszczególnych elementów

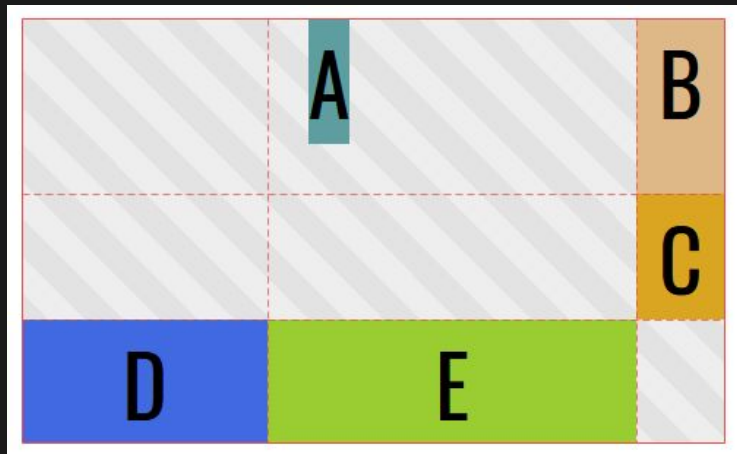
```
.grid {  
  display: grid;  
  grid-template-rows: 100px auto 70px;  
  grid-template-columns: 140px 1fr 50px;  
  // justify-items: stretch;  
  // align-items: stretch;  
}
```

```
.e1 {  
  grid-area: 1/1/3/3;  
  align-self: start;  
}
```



## align-self i justify-self - dla poszczególnych elementów

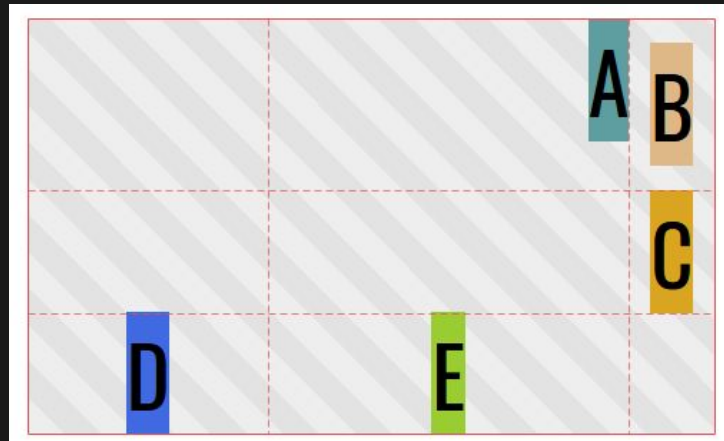
```
.grid {  
  display: grid;  
  grid-template-rows: 100px auto 70px;  
  grid-template-columns: 140px 1fr 50px;  
  // justify-items: stretch;  
  // align-items: stretch;  
}  
  
.e1 {  
  grid-area: 1/1/3/3;  
  align-self: start;  
  justify-self: center;  
}
```



align-self i justify-self - nadpisuje właściwości justify-items i align-items dla danego elementu

```
.grid {  
  display: grid;  
  grid-template-rows: 100px auto 70px;  
  grid-template-columns: 140px 1fr 50px;  
  place-items: center;  
}
```

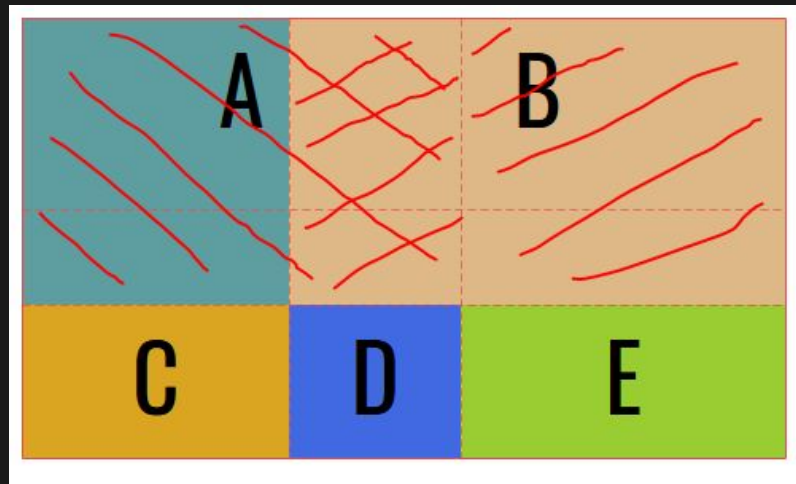
```
.e1 {  
  grid-area: 1/1/3/3;  
  align-self: start;  
  justify-self: right;  
}
```



# kolejność warstw - element na elemencie

```
.grid {  
  display: grid;  
  grid-template-columns: 140px 90px 1fr;  
  grid-template-rows: 100px 50px 80px;  
}
```

```
.e1 {  
  grid-area: 1/1/3/3;  
}  
.e2 {  
  grid-area: 1/2/span 2/span 2;  
}
```



Jeśli elementy nachodzą na siebie w komórce, to na wyższej warstwie (widocznej) jest ten który występuje później w HTML. W naszym przykładzie jest to element B, który nachodzi na element A.

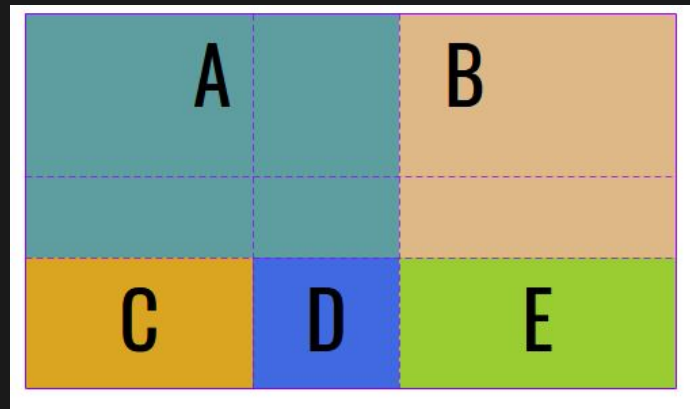


# z-index - kolejność na warstwach - zmiana

```
.grid {  
  display: grid;  
  grid-template-columns: 140px 90px 1fr;  
  grid-template-rows: 100px 50px 80px;  
}
```

```
.e1 {  
  grid-area: 1/1/3/3;  
  z-index: 1;  
}
```

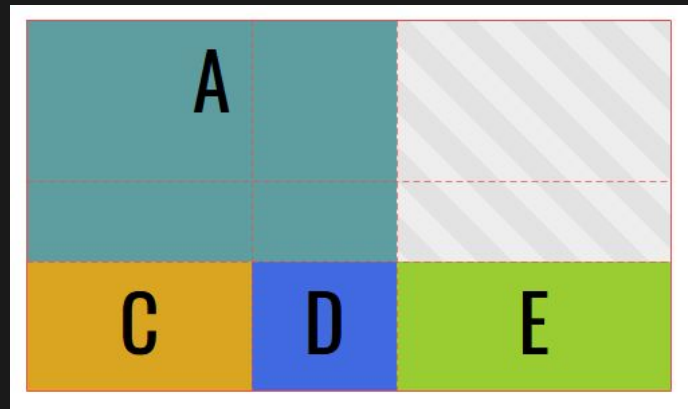
```
.e2 {  
  grid-area: 1/2/span 2/span 2;  
}
```





# z-index - kolejność na warstwach - zmiana

```
.grid {  
  display: grid;  
  grid-template-columns: 140px 90px 1fr;  
  grid-template-rows: 100px 50px 80px;  
}  
  
.e1 {  
  grid-area: 1/1/3/3;  
}  
  
.e2 {  
  grid-area: 1/2/span 2/span 2;  
  z-index: -1;  
}
```

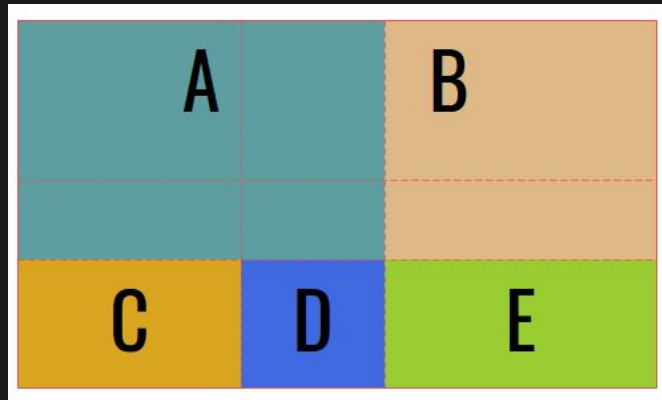


# order - kolejność na warstwach - zmiana

```
.grid {  
  display: grid;  
  grid-template-columns: 140px 90px 1fr;  
  grid-template-rows: 100px 50px 80px;  
}
```

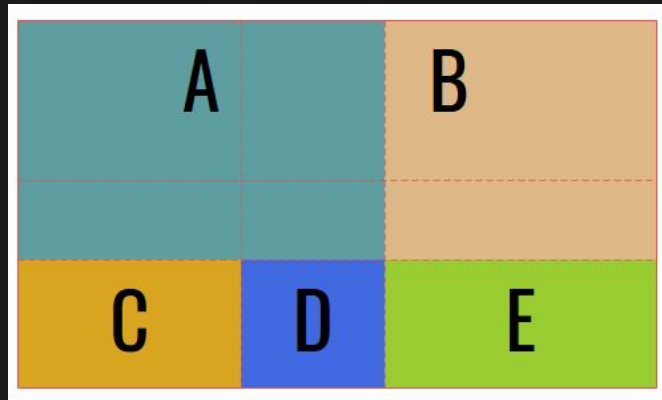
```
.e1 {  
  grid-area: 1/1/3/3;  
  order: 1;  
}
```

```
.e2 {  
  grid-area: 1/2/span 2/span 2;  
}
```



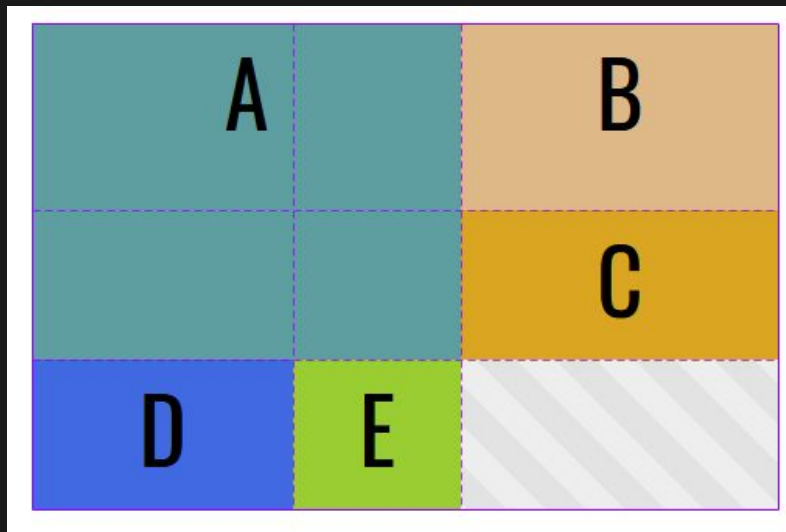
# order - kolejność na warstwach - zmiana

```
.grid {  
  display: grid;  
  grid-template-columns: 140px 90px 1fr;  
  grid-template-rows: 100px 50px 80px;  
}  
  
.e1 {  
  grid-area: 1/1/3/3;  
}  
  
.e2 {  
  grid-area: 1/2/span 2/span 2;  
  order: -1;  
}
```



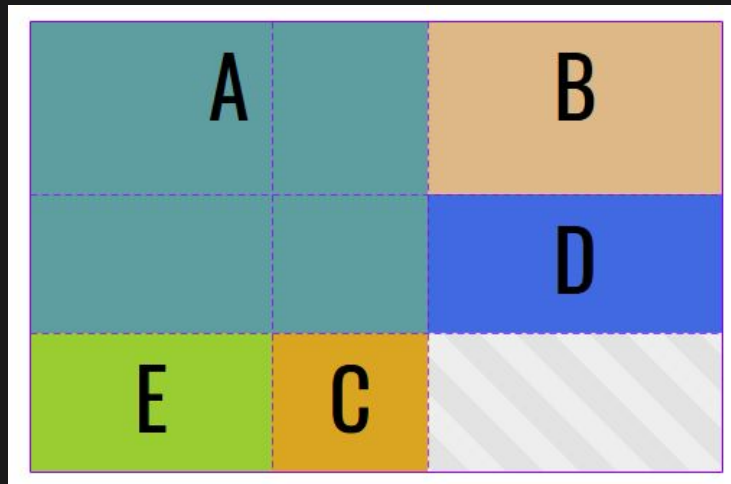
# Order - kolejność elementów, których pozycja generowana jest automatycznie (niejawnie)

```
.grid {  
  display: grid;  
  grid-template-columns: 140px 90px 1fr;  
  grid-template-rows: 100px 80px 80px;  
}  
  
.e1 {  
  grid-area: 1/1/3/3;  
}  
  
.e3 {  
}
```



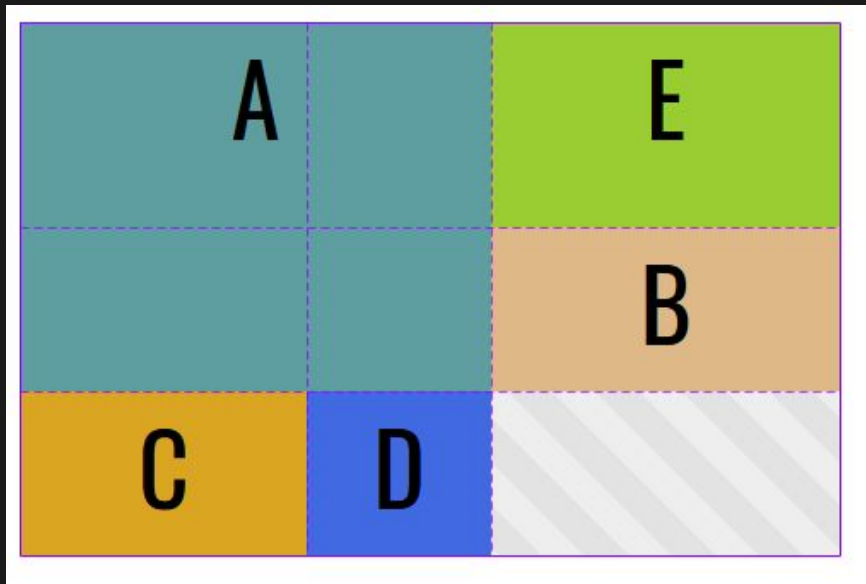
## Order - kolejność elementów, których pozycja generowana jest automatycznie (niejawnie)

```
.grid {  
  display: grid;  
  grid-template-columns: 140px 90px 1fr;  
  grid-template-rows: 100px 80px 80px;  
}  
  
.e1 {  
  grid-area: 1/1/3/3;  
}  
  
.e3 {  
  order: 1; //domyślnie elementy mają 0  
}
```



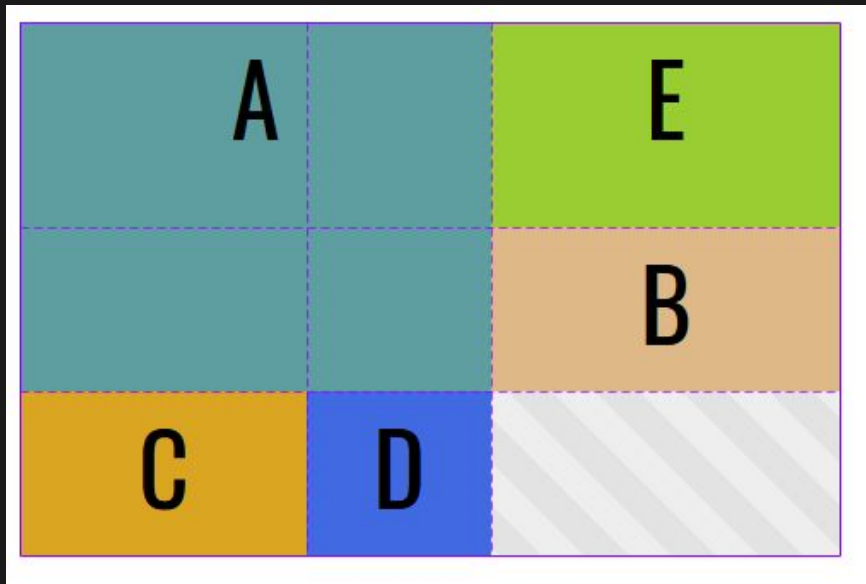
## Order - kolejność elementów, których pozycja generowana jest automatycznie (niejawnie)

```
.grid {  
  display: grid;  
  grid-template-columns: 140px 90px 1fr;  
  grid-template-rows: 100px 80px 80px;  
}  
  
.e1 {  
  grid-area: 1/1/3/3;  
}  
  
.e5 {  
  order: -1; //czym mniejszy order, tym wcześniej  
  //dany element zostanie umieszczony w siatce  
}
```



## Order - kolejność elementów, których pozycja generowana jest automatycznie (niejawnie)

```
.grid {  
  display: grid;  
  grid-template-columns: 140px 90px 1fr;  
  grid-template-rows: 100px 80px 80px;  
}  
  
.e1 {  
  grid-area: 1/1/3/3;  
  order: 5; //order nie wpływa na elementy jawnie  
            przypisane (wpływa oczywiście na kolejność na  
            warstwie)  
}  
  
.e5 { order: -1; }
```





# Zabawa z przykładami

- jednostki, wartości i funkcje CSS Grid





# Jednostka, funkcje i wartości Grid

**fr** // fraction (ułamek, frakcja). Jednostka, która w wersji minimalnej posiada wielkość zawartości elementu.

O ile w danym wymiarze (szerokość/wysokość) istnieje wolna przestrzeń, to jest ona dzielona między wszystkie kolumny/wiersze które posiadają wartość wyrażoną w jednostkach fraction.

**span** // wartość używana przy tworzeniu powierzchni dla elementu siatki np. `grid-column: 2/span 3;`

Oznacza to: polecenie: umieść element od 2 linii w 3 komórkach (można to też przeczytać od drugiej linii do linii przesuniętej o trzy kolejne czyli do piątej linii w tym wypadku).

**dense** // możemy użyć we właściwości `grid-auto-flow`. Wskazujemy, że jeśli istnieją wolne komórki przed aktualnym elementem CSS Grid powinien umieścić w nich element (domyślnie nie zostanie to zrobione). O co chodzi pokazywałem na przykładzie, więc jeśli nie pamiętasz, cofnij się do slajdu 48.



# Jednostka, funkcje i wartości Grid o których do tej pory nie mówiliśmy.

repeat()

minmax()

auto-fit

auto-fill

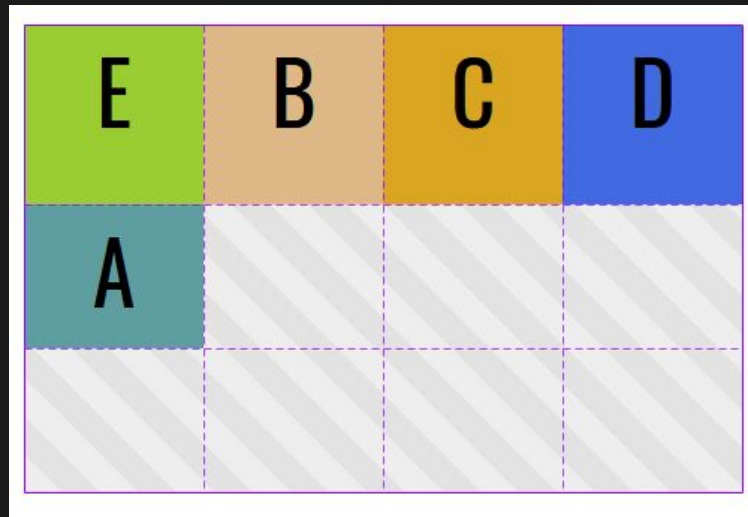
min-content

max-content



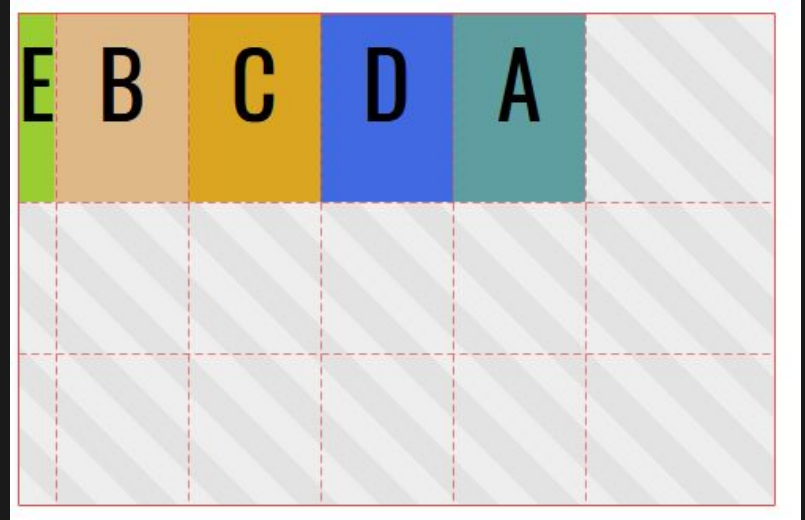
# funkcja repeat()

```
.grid {  
  display: grid;  
  grid-template-columns: repeat(4, 1fr);  
  grid-template-rows: 100px 80px 80px;  
}
```



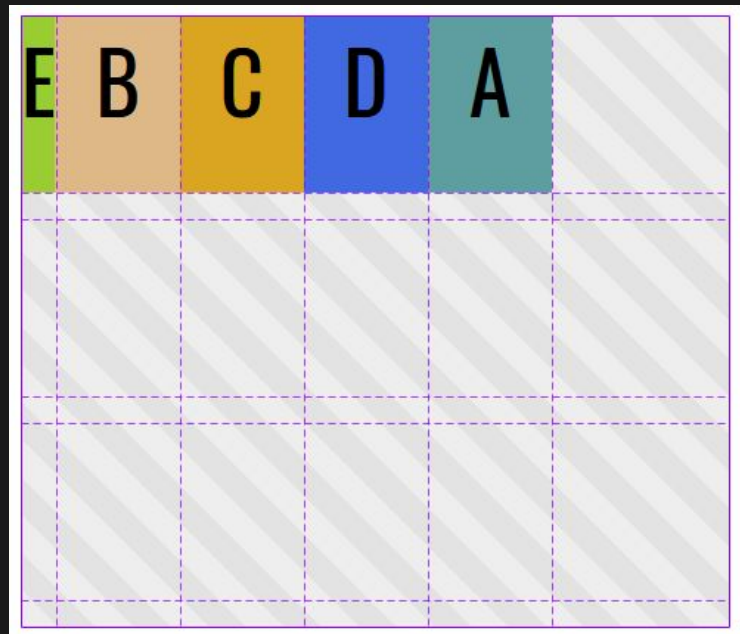
# funkcja repeat()

```
.grid {  
  display: grid;  
  grid-template-columns: auto repeat(4, 1fr) 100px;  
  grid-template-rows: 100px 80px 80px;  
}
```



# funkcja repeat()

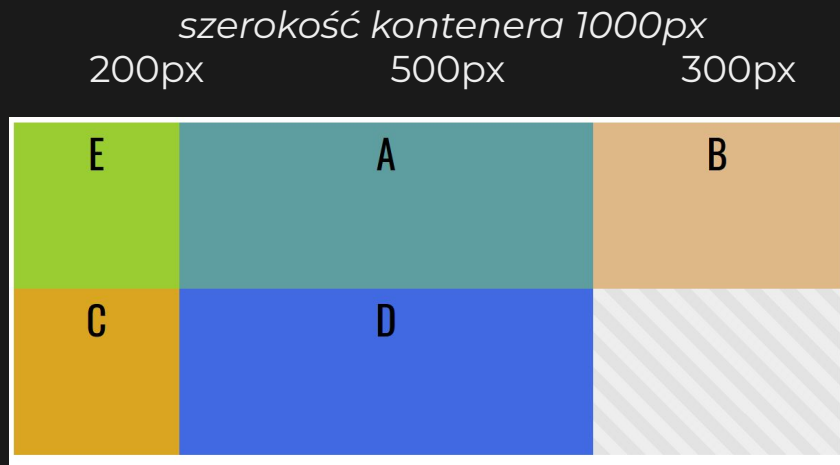
```
.grid {  
  display: grid;  
  grid-template-columns: auto repeat(4, 1fr) 100px;  
  grid-template-rows: repeat(3, 100px 15px);  
}
```



# funkcja minmax()

Możliwość określenia maksymalnej i minimalnej wielkości kolumny/wiersza.

```
.grid {  
  display: grid;  
  grid-template-columns: minmax(50px,200px) 1fr 300px;  
  grid-auto-rows: minmax(200px, auto);  
  width: 1000px;  
}
```

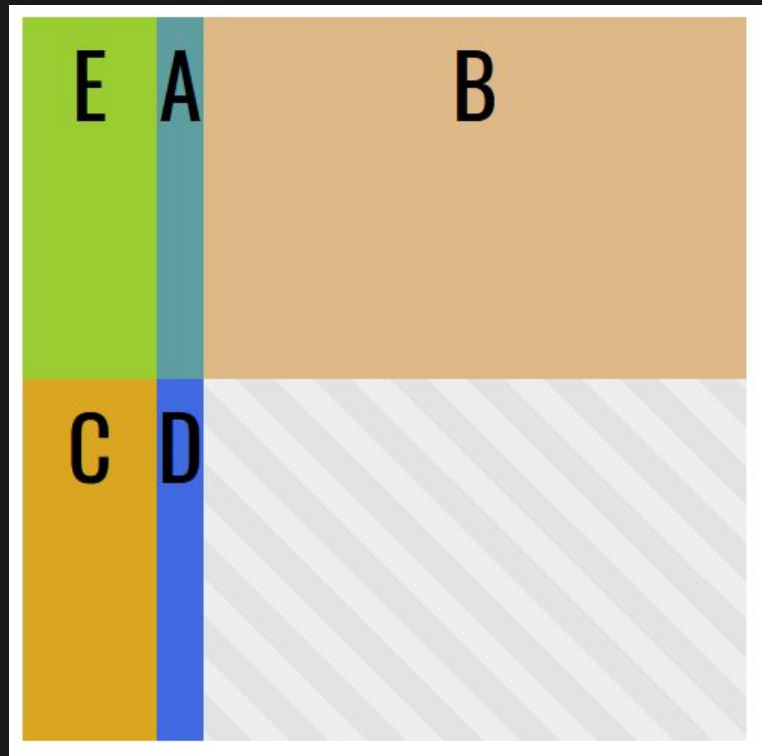


# funkcja minmax()

Frakcje (w zakresie wolnej przestrzeni) są wyliczane dopiero na końcu. Wcześniej swoją wielkość otrzyma więc wartość z funkcji minmax.

```
.grid {  
  display: grid;  
  grid-template-columns: minmax(50px,200px) 1fr 300px;  
  grid-auto-rows: minmax(200px, auto);  
  width: 400px;  
}
```

szerokość kontenera 400px  
75px 25px 300px



# wartości auto-fit i auto-fill

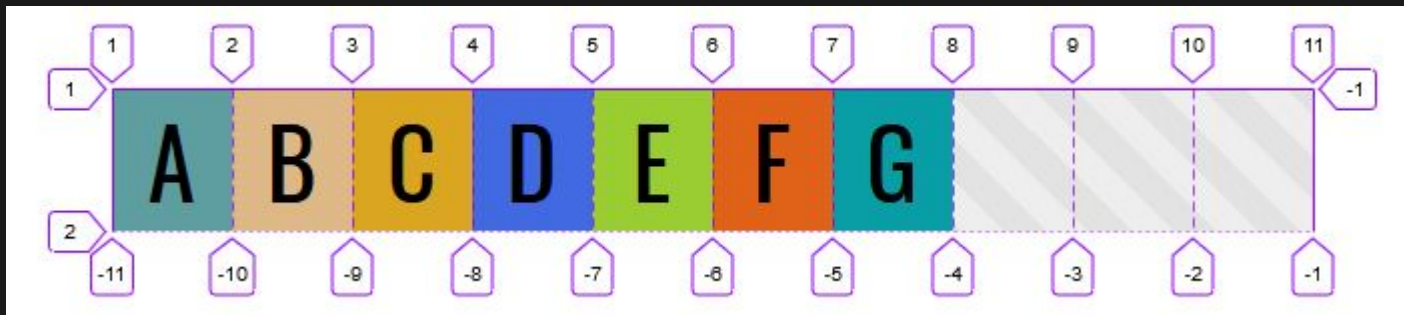
Do wykorzystania z funkcją repeat, jako alternatywa dla jawnego wskazania liczby wierszy/kolumn.

```
.grid {  
  display: grid;  
  grid-template-columns: repeat(auto-fit, minmax(55px, 1fr));  
}
```





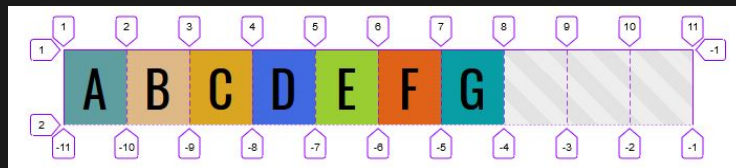
# wartość auto-fill



```
.grid {  
  display: grid;  
  grid-template-columns: repeat(auto-fill, minmax(55px, 1fr));  
  width: 600px;  
}
```



# wartość auto-fill



```
.grid {  
  display: grid;  
  grid-template-columns: repeat(auto-fill, minmax(55px, 1fr));  
  width: 600px;  
}
```

Tworzy maksymalną liczbę wierszy/kolumn jaka jest możliwa (bez względu na ilość elementów). W przykładzie jest w stanie utworzyć 10 kolumn po 55px (czyli 550px) - co wynika z wartości wielkości przekazanej jako drugi argument funkcji repeat (u nas drugi argument to minmax). Pozostała wolna przestrzeń jest przekazana do wszystkich kolumn (50px wolnej przestrzeni na 10 kolumn, co powoduje, że każda z nich zwiększa się o 5px, czyli ma w sumie 60px). Taką wielkość uzyskują nawet puste kolumny w naszym przykładzie.



# wartość auto-fit



```
.grid {  
  display: grid;  
  grid-template-columns: repeat(auto-fit, minmax(55px, 1fr));  
  width: 600px;  
}
```



# wartość auto-fit



```
.grid {  
  display: grid;  
  grid-template-columns: repeat(auto-fit, minmax(50px, 1fr));  
  width: 600px;  
}
```

Tworzy tyle kolumn ile elementów ma do rozdysponowania. W naszym przykładzie 7. Za sprawą funkcję minmax określamy tutaj też wielkość minimalną (w tym wypadku 50px) i maksymalną (1fr). W naszym przykładzie widzimy, że kontener ma wolne miejsce przy najmniejszej wartości (50px), więc każda kolumna uległa powiększeniu o wolną przestrzeń.



# wartość auto-fit

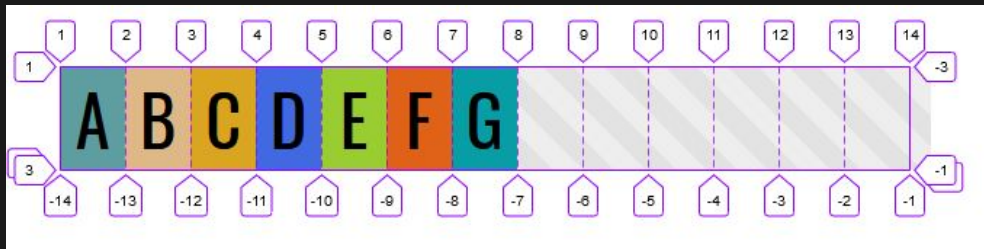


```
.grid {  
  display: grid;  
  grid-template-columns: repeat(auto-fit, 50px);  
  width: 600px;  
}
```

Tworzy tyle kolumn ile elementów ma do rozdysponowania. W tym przykładzie ustaliliśmy wielkość na sztywno. Widzimy więc, że siatka nie zajmuje całego kontenera.



# wartość auto-fill



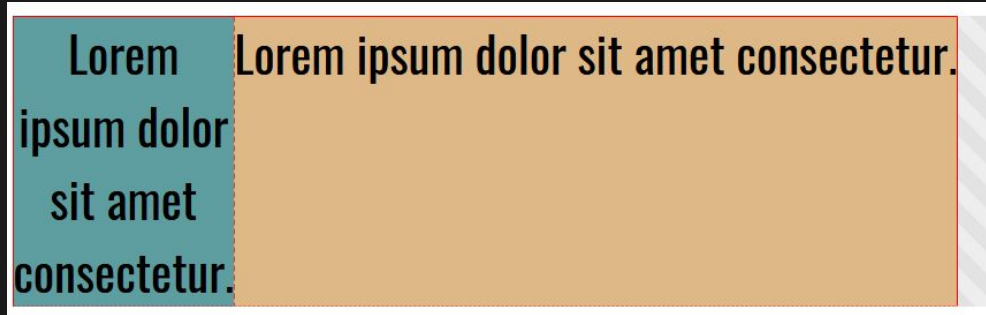
```
.grid {  
  display: grid;  
  grid-template-columns: repeat(auto-fill, 45px);  
  width: 600px;  
}
```

Tworzy tyle kolumn ile może, bez względu na ilość elementów. W tym przykładzie ustaliliśmy wielkość kolumny na sztywno. Ja widzimy w przykładzie, niekoniecznie cała siatka musiała zostać zajęta - zostanie ewentualnie tylko taka przestrzeń na której już zdefiniowana kolumna się nie zmieści. W naszym przykładzie udało się stworzyć 13 kolumn (585px) i pozostało 15px kontenera.



# min-content i max-content

```
.grid {  
  display: grid;  
  grid-template-columns: min-content max-content;  
}
```





Ufff to na tyle :)  
A teraz trochę praktyki

