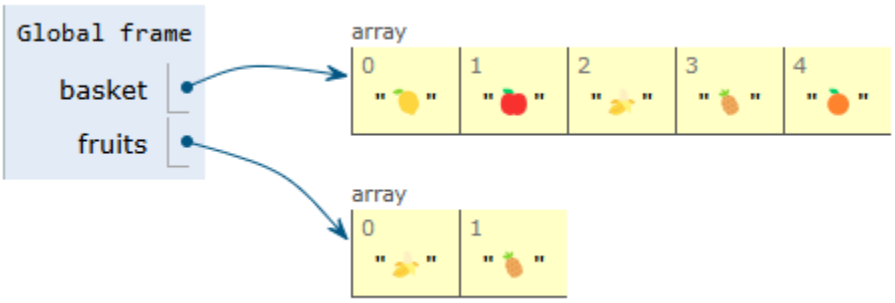
Prérequis

Commençons par l'étude de la méthode slice. Elle permet de découper une structure de données. La structure peut être un string ou un tableau.

La méthode est utilisée pour extraire une portion d'un tableau ou d'une chaîne **sans modifier l'original**. Elle prend deux arguments : l'indice de début (inclus) et, optionnellement, l'indice de fin (exclu).

Exemple :

1. const basket = ['🍋', '🍎','🍌','🍍','🍊'];
2. const fruits = basket.slice(2, 4); // ['banane', 'ananas']



Il est important de vérifier que slice laisse la structure de base intacte !

code

En action

Nous allons dans les cas des permutations utiliser slice ! Mais pourquoi est comment ?

Etude du code

```
1. let str = "ABCD";
2.
3. for (let i=0; i<str.length; i++){
4.    console.log(`c[${i}]=${str.slice(i,i+1)} : reste=${str.slice(0,i)+str.slice(i+1)}`)
5. }
```

code

str.slice(i,i+1) extrait un caractère individuel à l'index i. Notez que str n'est pas modifié.

str.slice(0,i)+str.slice(i+1) reconstruit la chaîne en omettant le caractère à l'index i . La construction se fait en concaténant deux morceaux de chaîne.

🚀Learn more

⛓Programme récursif !

Je rappelle que la programmation récursive est un fondement de la programmation.

Une fonction récursive est une fonction qui s'appelle elle-même.

🚀Exemple

L'exemple fil rouge reste le factoriel[1].

L'expression du factorielle s'écrit en langage mathématique :

---

[1] Il peut être difficile à appréhender.

$$n! = n * (n-1)!$$

😈Attention, Si nous écrivons le code suivant dans pythontutor, nous obtenons un message **"unknown crash due to server overload or bugs".**

1. const fact = (n) => n * fact (n-1);
2. fact(3);

Dans le code précédent, la fonction s'appelle tant qu'une condition stoppe l'appel. Ici, il n'y a pas de condition d'arrêt. Ceci est un problème.

🥷Condition d'arrêt

Une fonction récursive doit avoir une condition d'arrêt.

🪛Exemple

L'écriture de la fonction factorielle **fact.js** doit elle aussi avoir une condition d'arrêt. Voici son écriture en .

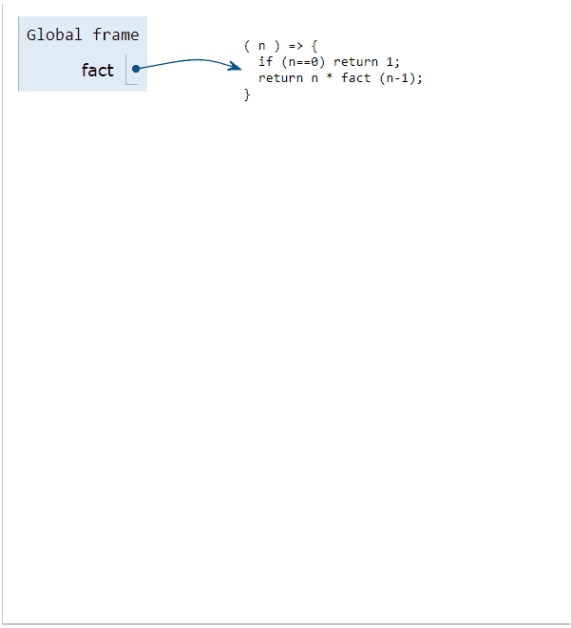
fact.js.

```
1. const fact = ( n ) => {
2.   if (n==0) return 1;
3.   return n * fact (n-1);
4. }
5.
6. const result = fact(3);
```

Lig.2 : Le test == de la valeur de n arrête les appels successifs avec return.

Pour mieux appréhender le comportement des différents appels, pythontutor reste un outil indispensable. Écrivons le code dans cet outil et regardons les appels de fonction dans cette animation.

Voici en animation l'empilement :



Chaque appel s'empile dans la pile des appels. Autrement dit: fact(n) attend le résultat de fact(n-1) qui attend le résultat de fact(n-2) …

C'est grâce à la condition d'arrêt 0! = 1 que l'empilement peut stopper. En effet, les appels s'arrêtent.

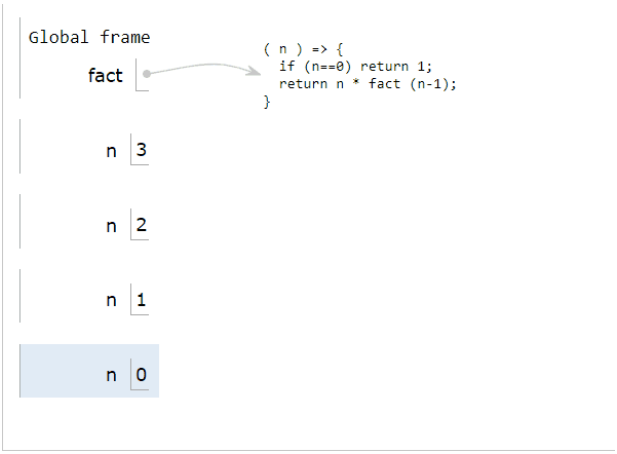
Il est donc possible de dépiler les appels car les fonctions peuvent maintenant rendre leur valeur.

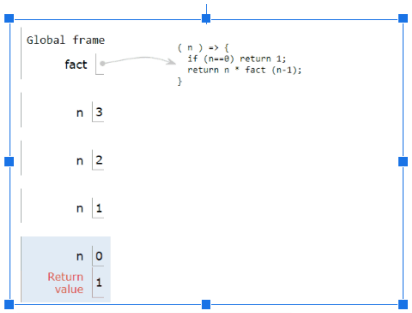
Ainsi 0! rend 1,

puis 1! rend 1*1,

puis 2! rend 2*2 …

Voici en action, le dépilement





Nous voyons l'importance du mot clé return qui permet de retourner la valeur du niveau inférieur (i-1) vers le niveau supérieur i des appels.

SOS Pour mieux comprendre

Voici un code Lien qui permet d'afficher des messages d'appel et de fin d'appel.

Nous comprenons que fact(3) attend le retour de fact(2) très longtemps. En effet, fact(2) attend également le retour de fact(1) qui …

Permutations p.7

```
APPEL    fact(3)
APPEL            fact(2)
APPEL                    fact(1)
APPEL                            fact(0)
FIN                      fact(1)
FIN              fact(2)
FIN      fact(3)
```

Pour renforcer la lisibilité, passons à une réécriture du code de base avec une mise en avant des blocs logiques.

🥷Réécriture

- Chaque bloc logique est bien séparé visuellement pour renforcer la lisibilité.
- La condition d'arrêt est maintenant clairement une "Étape 1" distincte et mise en avant.
- La fonction interne porte le nom **recursiveCall** , pour refléter son rôle dans les appels récursifs.

```
1. const factorial = (n) => {
2.   const recursiveCall = (currentValue) => {
3.     // Étape 1 : Vérifier la condition d'arrêt
```

```
    if (currentValue === 0) {
      return 1; // Retourner 1 car 0! = 1
    }

    // Étape 2 : Continuer l'appel récursif
    return currentValue * recursiveCall(currentValue - 1);
  };

  // Démarrer la récursivité
  return recursiveCall(n);
};

const result = factorial(3); // Résultat attendu : 6
console.log(result);
```

🥷Notez un terme technique : une closure !

Nous avons créé une **closure**, la fonction interne **recursiveCall** est définie à l'intérieur de **factorial**, permettant de conserver un espace lexical isolé.[2]

Exemple

Voici un exemple simple illustrant une **closure** et l'idée d'un espace lexical isolé :

```
function createCounter() {
```

[2] Une closure encapsule un espace lexical isolé, permettant à une fonction de "se souvenir" de son contexte, même si elle est exécutée en dehors de celui-ci.

```
2.    let count = 0; // Espace lexical isolé : cette variable est privée à
    createCounter
3.
4.    return function () {
5.      count++; // La fonction interne "se souvient" de count
6.      return count; // Retourne la valeur actuelle de count
7.    };
8.  }
9.
10. const counter = createCounter(); // Création de la closure
11. console.log(counter()); // Affiche : 1
12. console.log(counter()); // Affiche : 2
13. console.log(counter()); // Affiche : 3
```

Code

🥷Espace lexical isolé :

- La variable count est définie à l'intérieur de **createCounter**, donc elle n'est accessible que par la fonction retournée. Elle est "privée" au reste du code.
- La fonction retournée "se souvient" de son contexte (variable count) chaque fois qu'elle est appelée, même si **createCounter** a terminé son exécution[3].

[3] Lorsqu'une fonction a fini son exécution, toute définition de variable interne disparaît. mias pas ici !

- Ici, chaque appel à `counter()` incrémente et retourne la valeur de `count`, montrant comment une closure peut maintenir son propre état.

C'est une manière élégante d'encapsuler des données et de créer un comportement persistant en JavaScript. Fascinant, non ? 😊

Les permutations

Je vous laisse étudier le code des permutations !

```
1.  const generatePermutations = (chars, permutations = []) => {
2.
3.    const recursiveCall = (remaining, combination = "") => {
4.      // Condition d'arrêt : si rien ne reste, ajouter la combinaison finale
5.      if (!remaining) {
6.        permutations.push(combination); // Ajout de la permutation trouvée
7.        return; // Fin de la récursion
8.      }
9.
10.     // Générer les combinaisons pour chaque caractère restant
11.     for (let i = 0; i < remaining.length; i++) {
12.       const nextCombination = combination + remaining[i];
13.       const nextRemaining = remaining.slice(0, i) + remaining.slice(i + 1);
14.       recursiveCall(nextRemaining, nextCombination);
15.     }
16.  };
17.
```

```
recursiveCall(chars);
return permutations;
};

console.log(generatePermutations("ABC"));
```

🆘 à vous de jouer !

Et commencez à repérer la condition d'arrêt et l'appel récursif.

```
const generatePermutations = (chars, permutations = []) => {

  const recursiveCall = (remaining, combination = "") => {
    // Condition d'arrêt : si rien ne reste, ajouter la combinaison finale
    if (!remaining) {
      permutations.push(combination); // Ajout de la permutation trouvée
      return; // Fin de la récursion
    }


  // Générer les combinaisons pour chaque caractère restant
  for (let i = 0; i < remaining.length; i++) {
    const nextCombination = combination + remaining[i];
    const nextRemaining = remaining.slice(0, i) + remaining.slice(i + 1);
    recursiveCall(nextRemaining, nextCombination);
  }
};
```

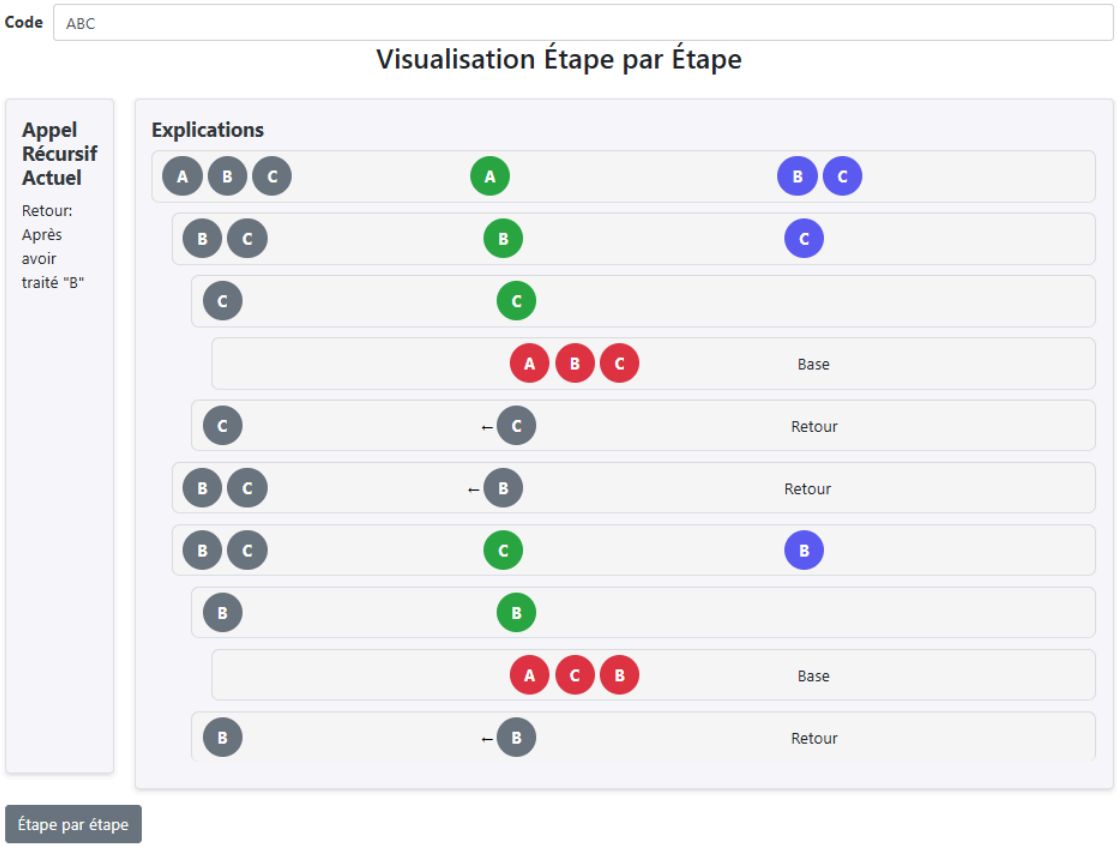
Permutations p.12

```
18.
19.  recursiveCall(chars);
20.  return permutations;
21. };
22.
23. console.log(generatePermutations("ABC"));
```

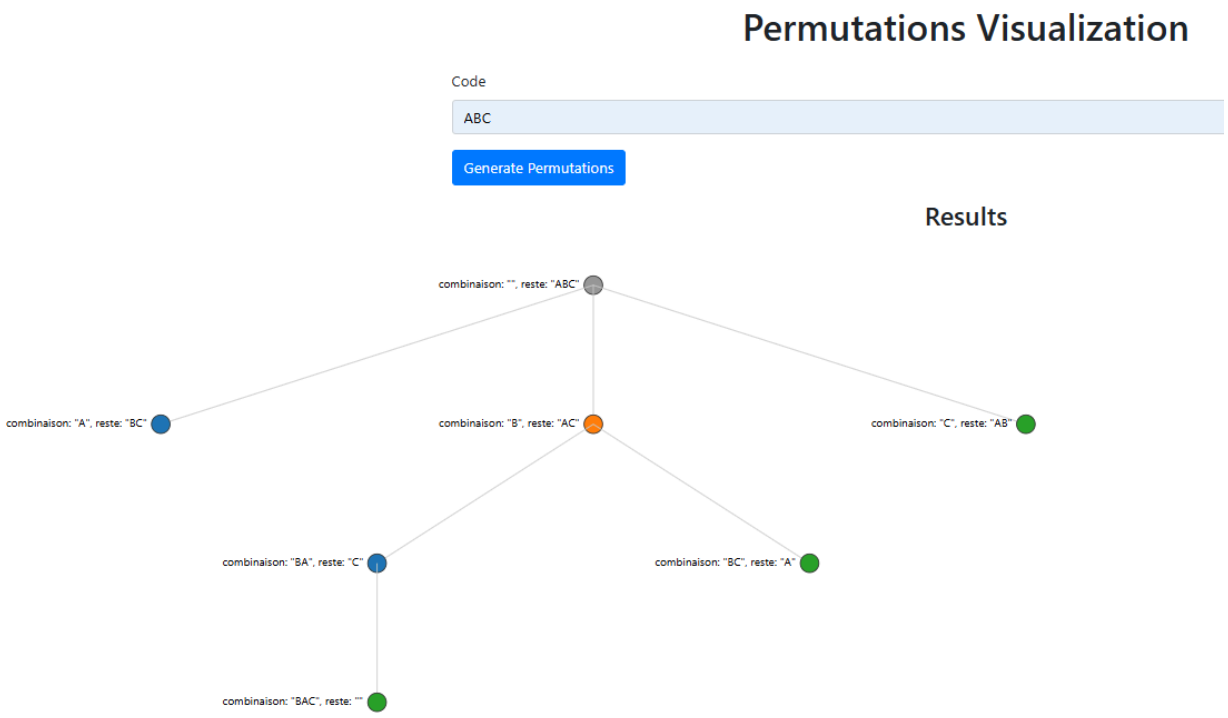
code

Des aides visuels :

https://dupontdenis.github.io/PermutationStepByStep/



https://dupontdenis.github.io/D3permut/


Permutations Visualization
Code
ABC
Generate Permutations
Results
combinaison: "", reste: "ABC"
combinaison: "A", reste: "BC"
combinaison: "B", reste: "AC"
combinaison: "C", reste: "AB"
combinaison: "BA", reste: "C"
combinaison: "BC", reste: "A"
combinaison: "BAC", reste: ""