Exercice 2

1. UseMemo et useCallback sont des hooks utilisés dans le but d'optimiser les performances par le biais de la mémorisation des valeurs ou des fonctions entre les rendus. Ils se différencient par le fait que UseMemo mémorise le résultat d'une fonction pour éviter des recalculs coûteux lors des rendus successifs, tandis que useCallback mémorise la définition d'une fonction pour éviter sa recréation à chaque rendu, ce qui est utile lorsqu'elle est passée en prop à un composant enfant optimisé.

2. Pour gérer le rendu conditionnel de composants lourds sans bloquer l'interface, utilisez le chargement différé avec React.lazy() et Suspense. Cela permet de charger les composants uniquement lorsqu'ils sont nécessaires, réduisant ainsi le temps de chargement initial et améliorant la réactivité de l'application. En enveloppant les composants chargés paresseusement dans un composant Suspense, vous pouvez afficher un indicateur de chargement pendant que le composant est en cours de chargement, évitant ainsi de bloquer l'interface utilisateur.

3. En React, la propriété key permet d'identifier de manière unique chaque élément d'une liste rendue avec map(). Elle aide React à optimiser les mises à jour en sachant précisément quels éléments ont changé, ont été ajoutés ou supprimés. Sans key, React peut avoir des difficultés à gérer efficacement les rendus, ce qui peut entraîner des comportements inattendus et des problèmes de performance.

4. Redux est plus adapté aux applications complexes avec un état global partagé et des interactions fréquentes entre composants. Pour des projets plus simples, des solutions plus légères et intégrées à React sont souvent suffisantes. **Applications simples ou petites** : Utiliser useState, useReducer ou Context API.

   - **Gestion d'état côté serveur** : Privilégier Tanstack Query ou Apollo Client.
   - **État localisé ou spécifique** : Recourir à useState ou useReducer dans les composants concernés.
   - **Optimisation des performances** : Opter pour des solutions légères comme Zustand ou Jotai.
   - **Projets à faible complexité ou durée limitée** : Éviter la complexité de Redux.

5. En React, la composition de composants est la méthode privilégiée pour réutiliser du code et construire des interfaces. Contrairement à l'héritage, la composition consiste à combiner plusieurs composants pour créer des interfaces complexes. L'héritage, bien que possible en JavaScript, est rarement utilisé en React car il est moins flexible et plus difficile à maintenir. Ci-dessous, des exemples :
    - Exemple d'héritage :

```
// Composant parent
class Button extends React.Component {
  render() {
    return <button>{this.props.label}</button>;
  }
}

// Composant enfant qui hérite de Button
class DangerButton extends Button {
  render() {
    return <button style={{ color: 'red' }}>{this.props.label}</button>;
  }
}
```

Inconvénient : ce modèle d'héritage rigidifie la structure, complique la maintenance et limite la flexibilité

- Exemple de composition :

```
// Composant Button réutilisable
const Button = ({ children, style, onClick }) => (
  <button style={style} onClick={onClick}>
    {children}
  </button>
);

// Composant DangerButton compose Button avec un style spécifique
const DangerButton = (props) => (
  <Button {...props} style={{ color: 'red', ...props.style }}>
    {props.children}
  </Button>
);
```

Ici, DangerButton compose le composant Button en lui passant des props et du contenu. Cette approche est flexible, claire et facile à étendre.

6. React utilise un algorithme de réconciliation pour comparer l'ancien et le nouveau Virtual DOM, identifiant les différences (diffing) afin de mettre à jour uniquement les parties modifiées du DOM réel, optimisant ainsi les performances.

7. Un composant contrôlé gère sa valeur via l'état React (useState), offrant un contrôle total sur les données. Un composant non contrôlé laisse le DOM gérer sa valeur, et l'accès se fait via une référence (ref)

8. useEffect(() => {}, []) s'exécute une seule fois après le premier rendu du composant, agissant comme componentDidMount dans les classes. Il est essentiel de bien gérer les dépendances dans le tableau []. Omettre des variables nécessaires peut entraîner des comportements inattendus, comme l'utilisation de valeurs obsolètes, ou des boucles infinies si l'effet modifie une dépendance incluse dans le tableau. Pour éviter cela, il est recommandé d'utiliser la règle exhaustive-deps fournie par le module eslint-plugin-react-hooks, qui vous avertira si des dépendances sont mal spécifiées et vous suggérera un correctif.