Algorytmy rekurencyjne

Łukasz Konieczny | LK4

1. Omówienie rekurencji

Rekurencja występuje, kiedy funkcja wywołuje samą siebie w pętli, aż do osiągnięcia pewnego warunku kończącego, przerywającego cykl wywołań.

Rekurencja dzieli się na dwa typy - pośrednią i bezpośrednią.

Pośrednia polega na wywoływaniu funkcji „F2” przez funkcję „F1”, oraz „F1” przez funkcję „F2”.

W rekurencji bezpośredniej, funkcja „F1” wywoływałaby sama siebie, bez funkcji pośredniej.

Innym podziałem rekurencji jest rekurencja liniowa i drzewiasta:

W rek. liniowej, funkcja wywołuje samą siebie tylko raz, natomiast w drzewiastej kilkukrotnie.

Zaletą rekurencji jest krótszy i łatwiejszy do odczytania / zrozumienia kod, jednak posiada ona wielką wadę w postaci zajmowania nieporównywalnie większej ilości czasu i miejsca w pamięci nawet dla prostych algorytmów.

Rekurencja często stosowana jest w połączeniu z „metodą dziel i zwyciężaj”. Polega ona na podziale problemu lub danego zestawu danych na jak najmniejsze części, rozwiązanie zadania dla takiej najmniejszej cząstki a następnie złączenie otrzymanych wyników w celu otrzymania ostatecznego rozwiązania.

Przykładem metody „dziel i zwyciężaj” jest algorytm „Merge Sort”, który dzieli tablicę na pół do momentu, aż nie będzie się dało jej już bardziej podzielić, a następnie łączy każdą z takich połówek ze sobą w celu otrzymania posortowanej tablicy wejściowej.

2. Ciąg Fibonacciego

Jest to rekurencyjny ciąg liczb naturalnych, w którym każdy element (oprócz zerowego i pierwszego) jest sumą dwóch poprzednich.

$$F_n = \begin{cases} 0 & \text{dla } n = 0, \\ 1 & \text{dla } n = 1, \\ F_{n-1} + F_{n-2} & \text{dla } n > 1. \end{cases}$$

Pierwsze wyrazy ciągu:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181

Jego matematyczna definicja od razu nasuwa nam pomysł zaimplementowania tego ciągu w sposób rekurencyjny:

```
long long fibRek(int n){
    if (n == 0 || n == 1)
        return n;

    return fibRek(n - 1) + fibRek(n - 2);
}
```

Widzimy, że rekurencyjna funkcja na wyznaczenie n-tego wyrazu tego ciągu jest bardzo prosta. Jej warunkiem kończącym jest „n == 0 lub n == 1”, zwraca ona wtedy odpowiednią wartość zgodnie z definicją ciągu. Jeśli nie jest on spełniony, wyznacza ona rekurencyjnie sumę dwóch poprzednich elementów. Łatwo zauważyć, że jest to rekurencja drzewiasta, ponieważ na każde jedno wywołanie funkcji przypadają dwa kolejne.

Niestety, z powodu wysokiej złożoności obliczeniowej, funkcja ta nie będzie w stanie wygenerować nam w rozsądnym czasie wartości ciągu dla większych „n”, na moim komputerze przestawała ona dawać odpowiedzi dla n = 47 i wyżej.

```
Program do wyznaczania wartosci n-wyrazu ciagu Fibonacciego
Lukasz Konieczny | LK4 | LAB 6

Jesli chcesz skorzystac z rekurencji, wpisz 1: 1
Podaj n: 46
Wynik: 1836311903
Program wykonal sie w 12102.3 milisekundy
```

Alternatywnym podejściem do rozwiązania ciągu Fibonacciego jest sposób iteracyjny:

```
long long fibIter(int n){
    if (n == 0 || n == 1)
        return n;

    long long first = 0, second = 1, tmp;

    for (int i = 2; i <= n; i++){
        tmp = first + second;
        first = second;
        second = tmp;
    }

    return second;
}
```

Aby oszczędzić czas, jeśli n == 0 lub 1 od razu zwracamy wynik. Jeśli nie, przechodzimy do wykonania algorytmu:

W zmiennych „first” i „second” trzymamy poprzednie wartości ciągu. Następnie w pętli przy pomocy zmiennej pomocniczej „tmp” sumujemy je ze sobą, przesuwamy aktualną wartość „second” do „first” i zapisujemy obliczoną sumę w „second”.
Po wszystkich wykonaniach pętli, ostatnia obliczona wartość ciągu znajduje się w zmiennej „second” i tą zmienną zwracamy.

Sposób iteracyjny nie zajmuje dużej ilości pamięci ponieważ nie dorzuca do niej kolejnych i kolejnych wywołań funkcji, dzięki czemu jest w stanie obliczyć wartości ciągu nawet dla bardzo dużych „n” w krótkim czasie:

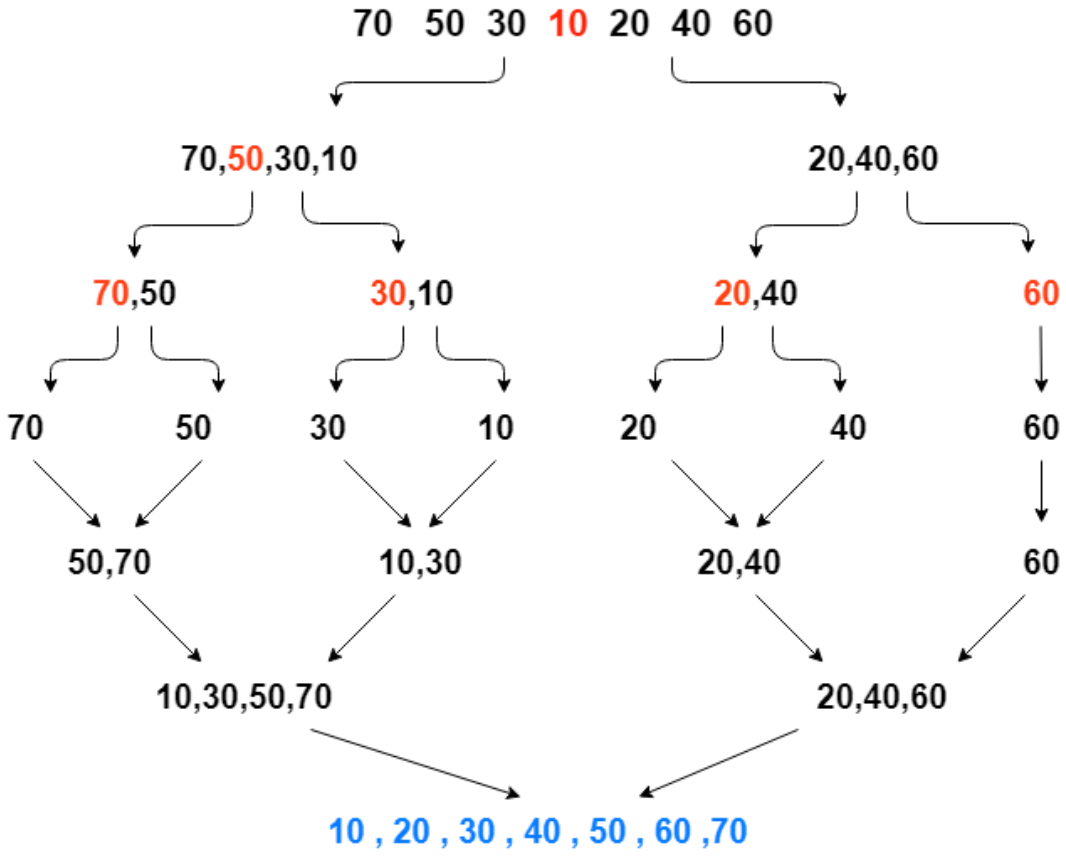
```
Program do wyznaczania wartosci n-wyrazu ciagu Fibonacciego
Lukasz Konieczny | LK4 | LAB 6

Jesli chcesz skorzystac z rekurencji, wpisz 1: 0
Podaj n: 400
Wynik: 2121778230729308891
Program wykonal sie w 0.0014 milisekundy
```

Jak widzimy, rekurencja, choć upraszcza zapis algorytmu, cechuje się bardzo słabą wydajnością, dlatego powinniśmy jej unikać jeśli tylko jesteśmy w stanie.

Rekurencyjna metoda obliczania wartości ciągu posiada złożoność O(2^n), ponieważ na każde wywołanie funkcji przypadają dwa kolejne, iteracyjna natomiast O(n) ponieważ szybkość jej wykonania zależy tylko od ilości wykonań pętli, czyli od „n”.

3. Merge Sort

Sortowanie przez scalanie (ang. „merge sort”) polega na rekurencyjnym podziale tablicy wejściowej na pół, do momentu w którym nie będziemy już mogli bardziej jej podzielić (każda połówka będzie pojedynczym elementem), a następnie złączeniem każdej z takich połówek w odpowiedniej kolejności elementów:



// Źródło: digitalocean.com

Merge sort również jest rekurencją drzewiastą, co ukazuje powyższy diagram.

```
void mergeSort(int* arr, int left, int right){
    if(left < right){
        int mid = (right + left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}
```

W każdym wywołaniu funkcji, jeśli warunek kończący nie jest spełniony (indeks startowy „left” jest mniejszy od końcowego „right”, czyli pomiędzy nimi są jeszcze elementy) wyznaczamy połowę zakresu na którym aktualnie pracujemy (elementy między „left” i „right”, inkluzywnie) a następnie wywołujemy rekurencyjnie funkcję dla obu wyznaczonych połówek.

Gdy sortowanie połówek się zakończy, łączymy je ze sobą przy pomocy funkcji „merge”:

```
void merge(int* arr, int left, int mid, int right){
    int maxIndex = right - left;
    int midIndex = mid - left;
    int* arrCopy = new int[maxIndex + 1];

    for(int i = 0; i <= maxIndex ; i++)
        arrCopy[i] = arr[left + i];

    int i = 0;
    int j = midIndex + 1 + i;
    int k = left;

    while(i <= midIndex && j <= maxIndex){
        if(arrCopy[i] <= arrCopy[j]){
            arr[k] = arrCopy[i];
            i++;
        }
        else {
            arr[k] = arrCopy[j];
            j++;
        }
        k++;
    }

    if(i == midIndex + 1){
        for(; j <= maxIndex; j++){
            arr[k] = arrCopy[j];
            k++;
        }
    }
    if(j == maxIndex + 1){
        for(; i <= midIndex; i++){
            arr[k] = arrCopy[i];
            k++;
        }
    }
}
```

Funkcja wykonuje kopię elementów w zakresie roboczym, aby uniknąć ich utraty podczas wstawiania posortowanych wartości do oryginalnej tablicy.

Zmienne „i” oraz „j” posłużą nam do iterowania po odpowiednio lewej i prawej połówce, a zmienna „k” do iterowania po oryginalnej tablicy.

Dopóki „i” ani „j” nie przekroczą wielkości swojej połówki, porównujemy ich elementy. Jeśli większy był element w połówce lewej, wstawiamy do

oryginalnej tablicy elementy z prawej połówki, dopóki nie napotkamy w niej elementu większego. Jeśli tak się stanie, wstawiamy elementy z lewej aż do napotkania sytuacji odwrotnej.

Po napotkaniu końca którejś z połówek, sprawdzamy czy któraś z nich dalej zawiera elementy, których nie wstawiliśmy. Jeśli tak, wstawiamy je wszystkie do oryginalnej tablicy (druga połówka w takiej sytuacji na pewno jest już „pusta", więc nie musimy wykonywać żadnego porównania).

W wyniku tych operacji, z dwóch połówek otrzymujemy jedną tablicę o posortowanych elementach.

Proces ten wykonywany jest dla wszystkich wyznaczonych rekurencyjnie połówek, w wyniku czego ostatnie scalenie da nam posortowaną tablicę wejściową.

Algorytm „merge sort" jest, pomimo wykorzystania rekurencji, jednym z najbardziej wydajnych algorytmów sortujących dzięki wykorzystaniu metody „dziel i zwyciężaj". Cechuje go złożoność O(nlogn).

4. Wnioski

Podczas zajęć zapoznaliśmy się z zaletami i wadami rekurencji, poznaliśmy jej rodzaje oraz metodę „dziel i zwyciężaj". Zaimplementowaliśmy jedne z najbardziej popularnych algorytmów rekurencyjnych, dzięki czemu zobaczyliśmy na własne oczy jak bardzo metoda iteracyjna potrafi być szybsza od rekurencyjnej, a jednocześnie jak metoda „dziel i zwyciężaj" potrafi uczynić rekurencję wydajną.