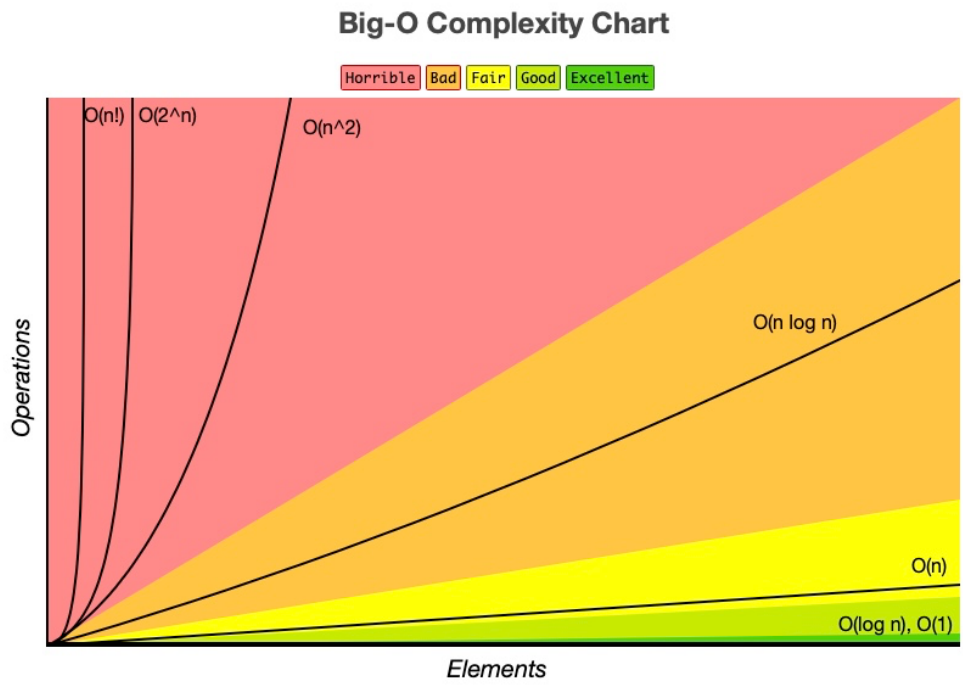
정렬(Sort)

정렬(Sorting)은 주어진 데이터를 일정한 기준에 따라 순서대로 나열하는 것을 말합니다. 정렬은 대부분 검색과 함께 사용되어 검색의 성능을 높이기 위해 사용됩니다.

시간 복잡도(time complexity)

시간 복잡도(time complexity)는 코드의 실행 시긴이 어떤 요인으로 결정되는지 나타내는 시간과 입력 데이터의 함수 관계입니다. 일반적으로 알고리즘의 시간 복잡도는 입력 크기에 대한 함수로 표현되며, 이를 Big O 표기법으로 나타냅니다.

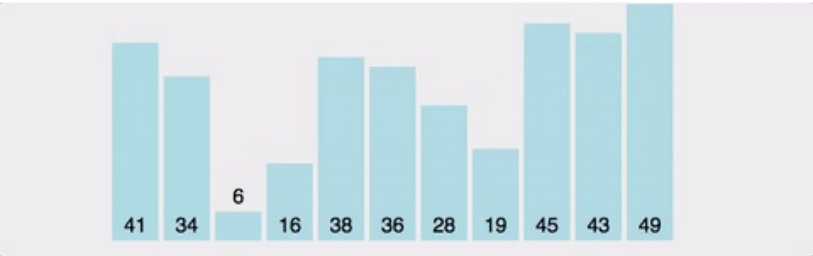
빅오 표기법(Big-O)



알고리즘 성능을 수학적으로 표기해주는 표기법이다. 알고리즘의 실행시간보다는 데이터나 사용자 증가률에 따른 알고리즘 성능을 예측하는게 목표이므로 중요하지 않은 부분인 상수와 같은 숫자는 모두 제거한다.

즉, 빅오 표기법은 불필요한 연산을 제거하여 알고리즘 분석을 쉽게 할 목적으로 사용된다.

거품 정렬(Bubble Sort)



Bubble Sort는 Selection Sort와 유사한 알고리즘으로 서로 인접한 두 원소의 대소를 비교하고, 조건에 맞지 않다면 자리를 교환하며 정렬하는 알고리즘 이다. 이름의 유래로는 정렬 과정에서 원소의 이동이 거품이 수면으로 올라오는 듯한 모습을 보이기 때문에 지어졌다고 한다.

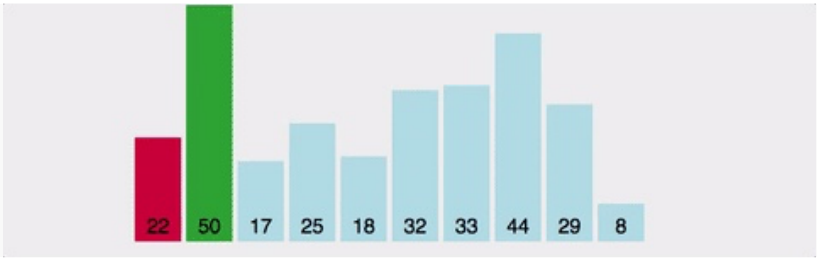
Bubble Sort Example (with Javascript)

```
function bubbleSort(arr) {
  var len = arr.length;
  for (var i = 0; i < len - 1; i++) {
    for (var j = 0; j < len - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        var temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
    }
  }
  return arr;
}

// 사용 예시
var arr = [41,34,6,16,38,36,28,19,45,43,49];
console.log("원래 배열:", arr);
console.log("정렬된 배열:", bubbleSort(arr));
```

버블 정렬의 시간 복잡도는 최선, 평균, 최악 모두 O(n^2) 입니다. 이는 배열의 길이가 길어질수록 비교와 교환 횟수가 많아져서 정렬 시간이 길어지기 때문입니다. 따라서 큰 배열에 대해서는 다른 정렬 알고리즘을 사용하는 것이 좋습니다.

선택 정렬(Selection Sort)



선택 정렬(Selection Sort)은 정렬되지 않은 배열에서 최솟값을 찾아 첫 번째 위치에 있는 값과 교환하고, 다음으로 작은 값을 찾아 두 번째 위치에 있는 값과 교환하고, 이 과정을 반복하여 정렬을 수행하는 알고리즘 입니다.

Selection Sort (with Javascript)

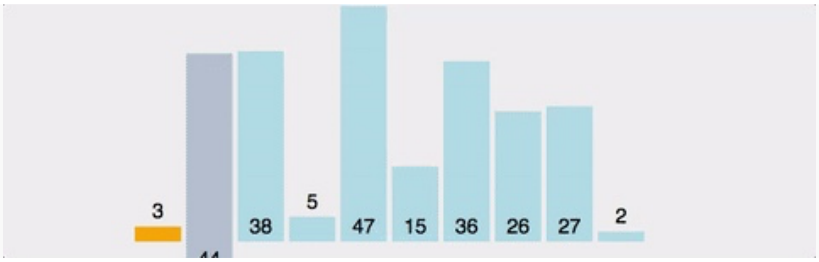
```
function selectionSort(arr) {
  const len = arr.length;
  let minIndex, temp;

  for (let i = 0; i < len - 1; i++) {
    minIndex = i;
    for (let j = i + 1; j < len; j++) {
      if (arr[j] < arr[minIndex]) {
        minIndex = j;
      }
    }
    temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
  return arr;
}

const arr = [22,50,17,25,18,32,33,44,29,8];
console.log("원래 배열:", arr);
console.log("정렬된 배열:", selectionSort(arr));
```

주어진 배열에서 최솟값을 찾아 교환하는 과정을 반복하여 정렬을 수행합니다. 최솟값을 찾기 위해 이중 반복문을 사용하며, 시간 복잡도는 O(n^2) 입니다.

삽입 정렬(Insertion Sort)



삽입 정렬(Insertion Sort)은 배열의 각 요소를 적절한 위치에 삽입하면서 정렬을 완성하는 알고리즘 입니다. 삽입 정렬은 선택 정렬과 유사하지만, 더 효율적인 알고리즘이며 작은 크기의 배열에서 사용됩니다.

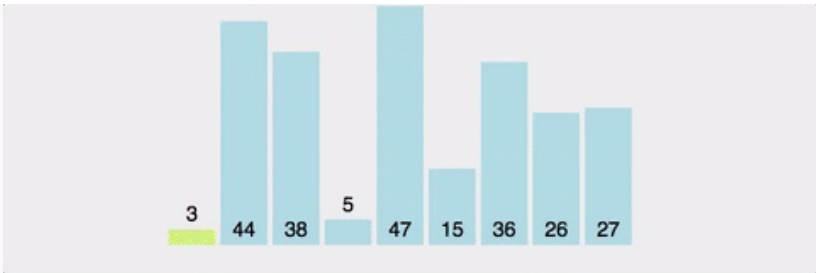
Insertion Sort (with Javascript)

```
function insertionSort(arr) {
  const n = arr.length;
  for (let i = 1; i < n; i++) {
    const current = arr[i];
    let j = i - 1;
    while (j >= 0 && arr[j] > current) {
      arr[j + 1] = arr[j];
      j--;
    }
    arr[j + 1] = current;
  }
  return arr;
}

const arr = [3,44,38,5,47,15,36,26,27,2];
console.log("원래 배열:", arr);
console.log("정렬된 배열:", insertionSort(arr));
```

시간 복잡도는 최선의 경우 O(n), 평균과 최악의 경우 O(n^2) 입니다. 따라서 삽입 정렬은 작은 크기의 배열에서 빠른 속도를 보이지만, 큰 크기의 배열에서는 느린 속도를 보입니다.

퀵 정렬(Quick Sort)



퀵 정렬(Quick Sort)은 분할 정복(Divide and Conquer) 알고리즘 중 하나로, 평균적으로 가장 빠른 정렬 알고리즘 중 하나입니다.

퀵 정렬은 다음과 같은 과정으로 이루어집니다.

1. 분할 (Partition) : 배열에서 하나의 원소를 pivot으로 선택한 뒤, pivot을 기준으로 작은 값은 왼쪽, 큰 값은 오른쪽으로 나눕니다.
2. 정복 (Conquer) : pivot을 제외한 왼쪽과 오른쪽 각각에 대해 분할을 수행합니다. 이 과정은 재귀적으로 수행됩니다.
3. 결합 (Combine) : 아무것도 수행하지 않습니다. 이미 왼쪽과 오른쪽이 정렬되어 있기 때문입니다.

이 과정을 통해 배열 전체가 정렬됩니다.

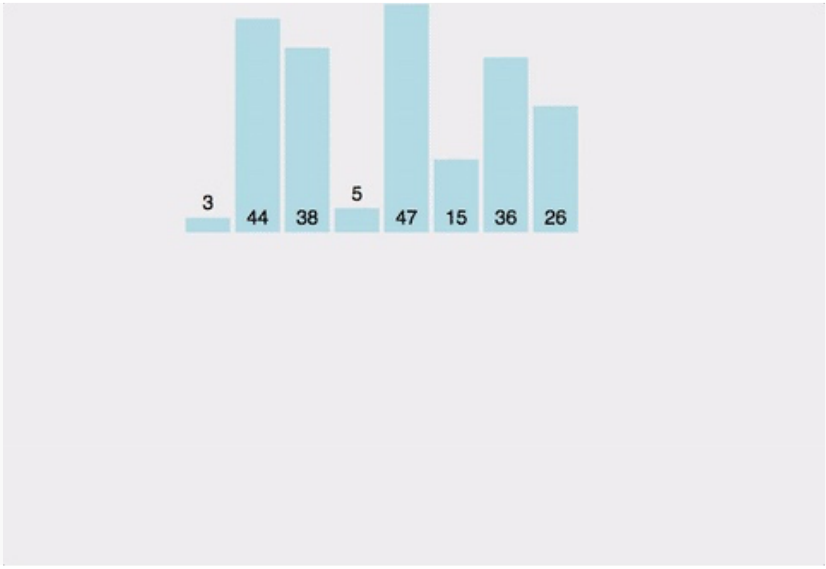
Quick Sort (with Javascript)

```
function quickSort(arr) {
  if (arr.length <= 1) {
    return arr;
  }
  const pivot = arr[0]; // 첫 번째 요소를 pivot으로 선택
  const left = [];
  const right = [];
  for (let i = 1; i < arr.length; i++) { // Partition
    if (arr[i] < pivot) { // pivot보다 작은 값은 왼쪽에 추가
      left.push(arr[i]);
    } else { // pivot보다 큰 값은 오른쪽에 추가
      right.push(arr[i]);
    }
  }
  // Conquer and Combine
  return quickSort(left).concat(pivot, quickSort(right));
}

const arr = [3,44,38,5,47,15,36,26,27];
console.log("원래 배열:", arr);
console.log("정렬된 배열:", insertionSort(arr));
```

퀵 정렬의 시간 복잡도는 평균적으로 O(nlogn) 입니다. 하지만 pivot이 항상 가장 작거나 큰 값일 경우 최악의 경우 O(n^2)의 시간 복잡도를 가지게 됩니다. 이러한 경우를 방지하기 위해 pivot을 선택하는 방법을 다양하게 사용합니다.

병합 정렬(Merge Sort)



병합 정렬(Merge Sort)은 분할 정복(Divide and Conquer) 알고리즘 중 하나로, 큰 문제를 작은 문제로 분해하고, 작은 문제를 해결해가면서 결과를 모아서 전체 문제를 해결하는 알고리즘입니다.

병합 정렬의 구현 방법은 다음과 같습니다.

1. 입력 배열을 두 개의 배열로 분할합니다.
2. 각각의 배열을 재귀적으로 병합 정렬합니다.
3. 두 개의 정렬된 배열을 하나의 배열로 병합합니다.

Merge Sort (with Javascript)

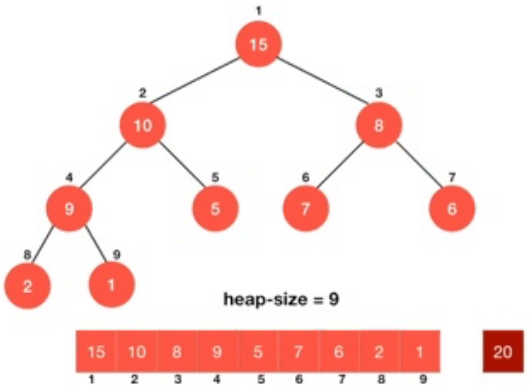
```
function mergeSort(arr) {
  if (arr.length <= 1) {
    return arr;
  }
  const mid = Math.floor(arr.length / 2);
  const left = arr.slice(0, mid);
  const right = arr.slice(mid);
  return merge(mergeSort(left), mergeSort(right));
}

function merge(left, right) {
  let result = [];
  while (left.length && right.length) {
    if (left[0] <= right[0]) {
      result.push(left.shift());
    } else {
      result.push(right.shift());
    }
  }
  while (left.length) {
    result.push(left.shift());
  }
  while (right.length) {
    result.push(right.shift());
  }
  return result;
}

const arr = [3,44,38,5,47,15,36,26];
console.log("원래 배열:", arr);
console.log("정렬된 배열:", mergeSort(arr));
```

병합 정렬의 시간 복잡도는 O(n log n)으로, 입력 배열의 크기가 커질수록 퀵 정렬보다 빠른 성능을 보입니다. 하지만 재귀호출로 인한 스택 메모리 사용량이 크다는 단점이 있습니다.

힙 정렬(Heap Sort)



힙 정렬(Heap Sort)은 힙(Heap) 자료구조를 이용한 정렬 알고리즘입니다. 힙이란 부모 노드와 자식 노드간의 대소 관계가 있는 트리 형태의 자료구조로, 최대 힙(Max Heap)과 최소 힙(Min Heap)으로 구분할 수 있습니다.

힙 정렬은 다음과 같은 과정을 거칩니다.

1. 주어진 배열을 최대 힙(Max Heap) 구조로 만듭니다.
2. 최대 힙에서 최대값을 꺼내 배열의 마지막 요소와 교환합니다.
3. 배열의 길이를 하나 감소시킨 후, 최대 힙으로 만듭니다.
4. 위 과정을 배열의 길이가 1이 될 때까지 반복합니다.

Heap Sort (with Javascript)

```javascript
function heapSort(arr) {
  // 배열의 길이가 1 이하면 정렬할 필요 없음
  if (arr.length <= 1) {
    return arr;
  }

  // Max Heap 구성
  for (let i = Math.floor(arr.length / 2) - 1; i >= 0; i--) {
    heapify(arr, arr.length, i);
  }

  // Max Heap 구성 후, root 값을 맨 끝 값과 교환하며 정렬 진행
  for (let i = arr.length - 1; i > 0; i--) {
    swap(arr, 0, i);
    heapify(arr, i, 0);
  }

  return arr;
}

// Max Heap 구성 함수
function heapify(arr, n, i) {
  let largest = i; // root 노드
  let left = 2 * i + 1;
  let right = 2 * i + 2;

  // 왼쪽 자식이 더 크다면, largest 인덱스 갱신
  if (left < n && arr[left] > arr[largest]) {
    largest = left;
  }

  // 오른쪽 자식이 더 크다면, largest 인덱스 갱신
  if (right < n && arr[right] > arr[largest]) {
    largest = right;
  }

  // root 노드가 자식보다 작다면, 자식과 교환하며 Max Heap 유지
  if (largest !== i) {
    swap(arr, i, largest);
    heapify(arr, n, largest);
  }
}

// 배열의 두 값을 교환하는 swap 함수
function swap(arr, i, j) {
  const temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}

// 테스트
const arr = [15,10,8,9,5,7,6,2,1,20];
console.log("원래 배열:", arr);
console.log("정렬된 배열:", heapSort(arr));
```

힙 정렬의 시간 복잡도는 O(n log n) 으로, 평균적으로 다른 비교 기반 정렬 알고리즘들과 같은 수준의 성능을 보입니다. 하지만 불균형한 데이터에서도 일정한 성능을 보이는 장점이 있습니다.

기수 정렬(Radix Sort)

기수 정렬(Radix Sort)은 비교 정렬 알고리즘이 아닌, 자리수를 기준으로 정렬하는 알고리즘이다. 예를 들어, 각 자리의 값이 0부터 9까지의 값을 가질 수 있는 4자리 수가 있다면, 먼저 1의 자리를 기준으로 정렬하고, 그 다음 10의 자리를 기준으로 정렬하고, 이렇게 4번째 자리까지 정렬한다.

Radix Sort (with Javascript)

```
function radixSort(arr) {
  const maxDigit = getMaxDigit(arr);
  let sortedArr = arr;
  for (let i = 0; i < maxDigit; i++) {
    sortedArr = countingSortForRadix(sortedArr, i);
  }
  return sortedArr;
}

// 최대 자리수 구하기
function getMaxDigit(arr) {
  let maxDigit = 0;
  for (let i = 0; i < arr.length; i++) {
    const numLength = Math.floor(Math.log10(arr[i])) + 1;
    maxDigit = Math.max(maxDigit, numLength);
  }
  return maxDigit;
}

// 기수에 해당하는 자리의 값을 기준으로 계수 정렬
function countingSortForRadix(arr, radix) {
  const count = Array.from({ length: 10 }, () => 0);
  const digitArr = [];
  const mod = Math.pow(10, radix + 1);
  const div = Math.pow(10, radix);

  for (let i = 0; i < arr.length; i++) {
    const num = Math.floor((arr[i] % mod) / div);
    count[num]++;
    digitArr.push(num);
  }

  for (let i = 1; i < count.length; i++) {
    count[i] += count[i - 1];
  }

  const resultArr = Array.from({ length: arr.length }, () => 0);
  for (let i = arr.length - 1; i >= 0; i--) {
    const num = digitArr[i];
    const index = count[num] - 1;
    resultArr[index] = arr[i];
    count[num]--;
  }

  return resultArr;
}

const arr = [15,10,8,9,5,7,6,2,1,20];
console.log("원래 배열:", arr);
console.log("정렬된 배열:", radSort(arr));
```

기수 정렬(Radix Sort)의 시간 복잡도는 O(d * (n + k))입니다. 여기서 d는 최대 자릿수, n은 배열의 크기, k는 진수(기수)입니다. 기수 정렬은 비교 정렬 알고리즘과는 달리 데이터를 비교하지 않고 자릿수를 기준으로 데이터를 정렬하기 때문에 성능이 좋습니다. 단, 데이터 크기가 커지면 자릿수가 많아져서 시간 복잡도가 높아질 수 있습니다.

계수 정렬(Count Sort)

계수 정렬(Counting Sort)은 정수나 정수로 표현할 수 있는 자료에 대해서만 사용할 수 있는 정렬 알고리즘입니다. 입력된 데이터와 관련된 특정한 값의 출현 회수를 세어(Counting) 그 숫자를 기반으로 데이터를 정렬하는 방법입니다.

계수 정렬의 구현 방법은 다음과 같습니다.

1. 입력값 중에서 최댓값을 구합니다.
2. 최댓값 크기의 배열을 생성합니다.
3. 입력값을 순회하며, 각 값이 몇 번 등장했는지 세어 배열에 저장합니다.
4. 배열에 저장된 값을 차례로 꺼내서 출력합니다.

Count Sort (with Javascript)

```
function countingSort(arr) {
  // 주어진 배열에서 최댓값을 찾습니다.
  const max = Math.max(...arr);

  // 0부터 최댓값까지 각 숫자가 나온 횟수를 저장할 배열을 만듭니다.
  const count = new Array(max + 1).fill(0);

  // 주어진 배열 내의 숫자들이 나온 횟수를 count 배열에 저장합니다.
  for (let i = 0; i < arr.length; i++) {
    count[arr[i]]++;
  }

  // count 배열을 누적합 배열로 만듭니다.
  for (let i = 1; i < count.length; i++) {
    count[i] += count[i - 1];
  }

  // 주어진 배열을 뒤에서부터 순회하며, count 배열에서 해당하는 인덱스를 찾아 정렬된 배열에 삽입합니다.
  const sortedArr = new Array(arr.length);
  for (let i = arr.length - 1; i >= 0; i--) {
    sortedArr[--count[arr[i]]] = arr[i];
  }

  return sortedArr;
}

// 테스트
const arr = [3, 5, 2, 8, 6, 4, 7, 1, 9, 0];
const sortedArr = countingSort(arr);
console.log(sortedArr); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

계수 정렬은 일반적으로 다른 정렬 알고리즘들보다 빠른 속도를 가지며, 최악의 경우에도 O(n + k)의 시간 복잡도를 가지기 때문에 매우 효율적인 알고리즘입니다. 단, 입력값의 크기가 매우 크거나 입력값이 연속적인 값을 가질 때는 메모리 사용량이 매우 커질 수 있으므로 유의해야 합니다.