히자료구조

1. 해시 함수 : 임의의 길이의 데이터를 고정된 길이의 데이터로 매핑하는 함수이다. 해시 함수에 의해 얻어지는 값은 해시 값, 해시 코드, 해시 라고 한다.

해시 테이블(자료구조) : 해시 맵 이라고도 한다. 컴퓨팅에서 키를 값에 매핑할 수 있는 구조이다.

1. 큐 : 큐는 컴퓨터의 기본적인 자료구조의 한가지로, 먼저 집어 넣은 데이터가 먼저 나오는 FIFO구조로 저장하는 형식을 말한다. 프린터의 출력이나 윈도 시스템의 메시지처리기, 프로세스 관리 등 데이터가 입력된 시간 순서대로 처리해야 할 필요가 있는 상황에 이용된다.
2. 스택 : 한 쪽 끝에서만 자료를 넣거나 뺄 수 있는 선형 구조 LIFO으로 되어 있다. 자료를 넣는 것을 push 꺼내는 것을 pop이라고 한다.
3. 트리 : 트리 구조는 그래프의 일종으로, 여러 노드가 한 노드를 가리킬 수 없는 구조이다. 회로가 없으며 서로 다른 두 노드를 잇는 길이 하나 뿐인 그래프를 트리라고 한다. 트리의 최상위 노드를 루트 노드, A가 노드 B를 가리키면 A는 부모 노드 B는 자식 노드 라고 한다. 자식 노드가 없는 노드를 leaf노드라고 한다. 또한 leaf 노드가 아닌 노드를 내부 노드라고 한다.
4. 우선순위 큐 : 들어간 순서와 상관 없이 우선순위가 높은 순으로 나온다. 우선순위 큐는 힙이라는 자료구조를 사용하여 구현할 수 있다. 힙은 완전 이진트리이다. 각 레벨별로 부모와 자식만 비교하면 된다.

알고리즘

1. 그리디 알고리즘(탐욕) : 최적해를 구하는 데에 사용되는 근사적인 방법으로, 여러 경우 중 하나를 결정해야 할 때마다 그 순간에 최적이라고 생각되는 것을 선택해 나가는 방식으로 진행하여 최종적인 해답에 도달한다. 그 순간에 대해 지역적으로는 최적이지만, 그 선택들을 계속 수집하여 전역적인 해답을 만들었다고 해서, 그것이 최적이라는 보장은 없다.

탐욕 알고리즘 문제는 탐욕스런 선택 조건과 최적 부분 조건이라는 두 가지 조건이 만족 된다. 탐욕스런 선택 조건은 앞의 선택이 이후의 선택에 영향을 주지 않는다. 최적 부분 조건은 문제에 대한 최적해가 부분 문제에 대해서도 역시 최적해라는 것이다. 대부분의 경우 계산 속도가 빨라서 실용적으로 사용 할 수 있다.

1. DP 알고리즘(동적 계획법) : 처음 진행되는 연산은 기록해 두고, 이미 진행했던 연산이라면 다시 연산하는 것이 아니라 기록되어 있는 값을 가져오는 것이다. 작은 문제가 반복적으로 일어나며 같은 문제는 구할 때마다 정답이 같아야 사용한다. 구현 방법으로는 bottom-up & top-down 방법이 있다 top-down은 재귀로 구현한다.
2. 깊이 우선 탐색: DFS는 탐색방법의 하나로 탐색트리의 최근에 첨가된 노드를 선택하고, 이 노드에 적용 가능한 동작자 중 하나를 적용하여 트리에 다음 수준의 한 개의 자식노드를 첨가하며, 첨가된 자식 노드가 목표노드 일 때까지 앞의 자식 노드의 첨가 과정을 반복 해 가는 방식이다.
3. 너비 우선 탐색 : BFS는 탐색방법의 하나로 시작 정점을 방문 후 시작 정점에 인접한 모든 정점들을 우선 방문하는 방법이다. 더 이상 방문하지 않은 정점이 없을 때까지 한다. 큐를 사용하여 구현한다.
4. 다익스트라 알고리즘 : 최단경로를 탐색하는 알고리즘이라고 볼 수 있다. Dp알고리즘이 라고도 볼 수 있다. 최단 거리는 여러 개의 최단 거리로 이루어져 있으므로 계속해서 갱신하여 사용한다. 모든 간선이 양수인 경우 and 우선순위 큐로 구현하면 빠름
5. 결정알고리즘 & 이분 검색 : 어떠한 범위에 무조건 답이 있다는 것을 알고 있을 때 범위를 좁혀 가며 탐색하는 알고리즘이다
6. 벨만 – 포드 알고리즘 : 단일 시작점 , 방향 그래프, 음수 가중치, 정점의 -1 만큼 반복하며 모든 간선들을 돌며 가중치를 수정한다. V – 1 번이 넘어가서도 가중치가 수정이 되면 음수 사이클이 존재 하는 것이다. 매번 간선들을 도는 것이 중요하다.
7. 크루스칼 알고리즘 : 간선을 오름차순으로 정렬한다. 간선을 하나씩 확인하며 사이클의 생성 유무를 판단한다. 최소 신장 트리 구하기 위해서 사용한다 -> prime(간선이 많을 때 사용한다) 알고리즘도 있다.
8. KMP 알고리즘 : 접두사와 접미사를 사용한 문자열 검색 :
9. *// parent: 전체 문자열, pattern: 찾고자하는 문자열*
10. public ArrayList<Integer> kmp(String parent, String pattern) {
11. *// 문자열을 찾은 인덱스를 저장할 list*
12. ArrayList<Integer> list = new ArrayList<>();
14. *// pi배열을 생성*
15. int[] pi = getPi(pattern);
16. int n = parent.length(), m = pattern.length();
17. int j = 0; *// i: 전체 문자열의 index, j: 패턴 문자열의 index*
18. for (int i = 0; i < n; i++) {
19. while (j > 0 && parent.charAt(i) != pattern.charAt(j))
20. j = pi[j - 1];
21. if (parent.charAt(i) == pattern.charAt(j)) {
22. *// m은 패턴의 길이, 길이와 j가 같다면 패턴을 모두 찾은 경우*
23. if (j == m - 1) {
25. *// 문자열에서 패턴이 시작한 인덱스를 계산해 list.add*
26. list.add(i - m + 1);
27. j = pi[j];
29. } else j++;
30. }
31. }
32. return list;
33. }
34. public int[] getPi(String pattern) {
35. int m = pattern.length();
36. int[] pi = new int[m];
37. int j = 0;
38. for (int i = 1; i < m; i++) {
39. while (j > 0 && pattern.charAt(i) != pattern.charAt(j))
40. j = pi[j - 1];
41. if (pattern.charAt(i) == pattern.charAt(j))
42. pi[i] = ++j;
43. }
44. return pi;
45. }
46. 위상 정렬 : 접두사와 접미사를 사용한 문자열 검색 :

시간복잡도 : 기본적인 연산을 수행하는 데에 어떤 고정된 시간이 걸릴 때, 알고리즘에 의해서 수행되는 기본 연산의 개수를 세어 예측할 수 있다. 그러므로 걸리는 시간의 총량과 알고리즘에 의해 수행되는 기본적인 연산의 개수는 최대 상수 인자 만큼 다르다.

정규 표현식 :

https://velog.io/@sa1341/Java-정규-표현식

자주 사용하는 메소드

Import java.util.Scanner

Next() : 공백 문자를 기준으로

NextLine() : 엔터를 기준으로 개행 문자

String

- toUpperCase()

- toLowerCase()

- trim() : 양쪽 공백 제거

- replace(char,char)

- replaceAll(String,String)

- replaceFisrt(String,String)

- split(String)

- split(String, int)

- substring(int,int)

- substring(int start)

Character

- toUpperCase()

- toLowerCase()

- isUpperCase()

- isLowerCase()

- isLetter()

- isDigit()

- getNumericValue()

StringBuilder

* append()
* reverse()
* toString()
* insert()
* delete(int,int)
* deleteCharAt(int)

Integer

* valueOf(String, rdx)

Import java.util.ArrayList

Import java.util.ArrayList

//중복 조합 (순서관심없고 뽑은 유무만 생각 + 자기자신도 포함)

**private** **static** **void** reCombination(**int**[] reComArr, **int** n, **int** r, **int** index, **int** target) {

**if**(r==0){

**for**(**int** i : reComArr){

                System.out.print(i+" ");

            }

            System.out.println();

**return**;

        }

**if**(target == n)**return**;

        reComArr[index] = target;

        reCombination(reComArr, n, r-1, index+1, target);//뽑는경우

        reCombination(reComArr, n, r, index, target+1);//안뽑는경우

    }

    //조합 (순서관심없고 뽑은 유무만 생각)

**private** **static** **void** combination(**int**[] comArr, **int** n, **int** r, **int** index, **int** target) {

**if**(r==0){

**for**(**int** i : comArr){

                System.out.print(i+" ");

            }

            System.out.println();

**return**;

        }

**if**(target == n)**return**;

        comArr[index] = target;

        combination(comArr, n, r-1, index+1, target+1);//뽑는경우

        combination(comArr, n, r, index, target+1);//안뽑는경우

    }

    //중복순열 (순서있게 배열 + 자시자신도 포함)

**private** **static** **void** rePermutation(**int** n, **int** r, LinkedList<Integer> rePerArr) {

**if**(rePerArr.size() == r){

**for**(**int** i : rePerArr){

                System.out.print(i+" ");

            }

            System.out.println();

**return**;

        }

**for**(**int** i=0; i<n; i++){

            rePerArr.add(i);

            rePermutation(n, r, rePerArr);

            rePerArr.removeLast();

        }

    }

    //순열 (순서있게 배열)

**private** **static** **void** permutation(**int** n, **int** r, LinkedList<Integer> perArr, **int**[] perCheck) {

**if**(perArr.size() == r){

**for**(**int** i : perArr){

                System.out.print(i+" ");

            }

            System.out.println();

**return**;

        }

**for**(**int** i=0; i<n; i++){

**if**(perCheck[i] == 0){

                perArr.add(i);

                perCheck[i] = 1;

                permutation(n, r, perArr, perCheck);

                perCheck[i] = 0;

                perArr.removeLast();

            }

        }

    }