

Radix Trie의 구현과 알고리즘 특성 분석

2021학년도 청명고등학교 2학년 수학과제탐구

2학년 2번 2번
고호정

서론

사실부터 말하자면 저는 알고리즘을 알지 못하는 사람입니다. 많은 사람들이 알고리즘을 수학적으로 접근하려고 시도하지만 저는 그것을 하나의 논리력으로 판단했기 때문에 꼭 수학이 필요할까라는 생각도 많이 했었습니다. 하지만 시간이 지나서 보니 우리가 논리를 표현하고 있는 방법 그 자체가 수학이었고 앞으로 무언가를 다른 사람들이랑 더욱 더 효율적으로 이루어 나가려면 수학이 필요하다고 느껴졌습니다. 그 외에도 개인적으로 개발을 진행하면서 수학으로 이득을 꽤 본 경험이 있기도 합니다. 이번 수학 과제 탐구 주제는 알고리즘을 기반으로 하여 결정되었습니다.

가장 먼저 찾아본 것은 Radix Trie입니다. Trie(이하 "Trie")가 쉬운 알고리즘은 아니지만 현재까지 제가 개발을 해오면서 내게 과연 알고리즘이 준 혜택 중에 무엇을 가장 크게 받았을까 생각을 해보았더니 Radix Trie가 떠올랐습니다. 저는 웹 쪽 기술에서 주로 활동하는 소프트웨어 엔지니어였는데 Fastify의 find-my-way라는 URL 핸들러를 Radix Trie로 기존 알고리즘보다 3배 이상 빠르다는 점에서 굉장히 흥미롭게 보았습니다. 또 이 때 처음으로 알고리즘에 대한 저의 시선이 '그저 어렵다'에서 '나도 해보고 싶다'로 바뀌었습니다.

Radix Trie란 무엇인가?

무언가를 하기 전에 일단 과연 무엇이 Radix Trie이고 발음이 비슷한 Tree 등의 알고리즘과의 차이점을 명확하게 할 필요가 있었습니다. 그리고 Radix Trie에 대해 알아보면서 그 전에 내가 미리 알아야 할 선행 지식은 무엇인가 또한 고려를 해보았습니다. 온라인 검색을 통해 앞으로 무엇을 목표로 할 지 고민했습니다. 가장 먼저 Trie(이하 "트라이")와 Tree(이하 "트리")의 차이에 대하여 정확히 하였는데 의외로 그 둘은 크게 다르지 않습니다. 기본적으로 각각의 알고리즘은 스스로를 대변하는 노드를 가지고 있습니다. 간단한 예를 들자면: radix는 트라이 구조에서 r-a-d-i-x로 나타낼 수 있습니다.

그런데 여기에서 계속 언급되고 있는 트라이는 문자열, 즉, 단어나 문장을 트리 구조로 나타낸 것입니다. 다른 알고리즘보다 트라이가 문자열 검색에서 더 빠른 성능을 보여주는 이유는 단순히 "수많은 집합"을 가지고 있기 때문입니다. 트라이는 하나의 노드가 또 다른 노드를 가리키고 그 길을 따라 찾아가는 과정으로 손쉽게 실제로 검색 시에 필요한 노드만 발췌할 수 있게 만들어져 있습니다.

Radix 트라이는 여기에서 정말로 우리가 원하는 부분만 발췌한 알고리즘이라고 할 수 있습니다. 위에서 radix를 r-a-d-i-x와 같이 나타내었는데 문장으로 같은 예시를 만들어보겠습니다. Hello world는 h-e-l-l-o-<공백>-w-o-r-l-d가 됩니다. 하지만 우리가 실제로 텍스트를 검색할 때는 키워드 위주, 즉, 단어 위주로 검색을 하기 때문에 한 글자마다 짜르는 것은 불필요하게 느껴집니다.

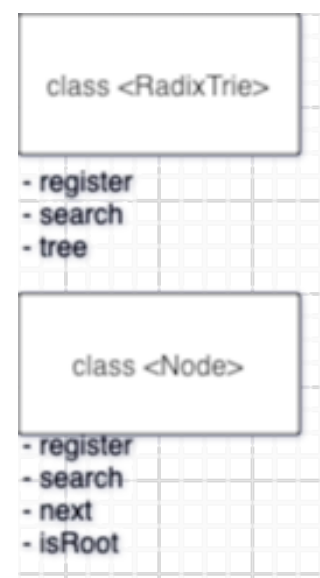
이러한 문제를 해결하기 위해 Radix 트라이가 등장했습니다. 트라이의 한 종류로 기존에는 한 글자를 하나의 노드(부분)으로 취급하였다면 이제는 하나의 단어를 노드의 부분으로 취급하는 것입니다. 즉, Hello world를 Hello-world와 같은 구조로 나타낼 수 있게 됩니다. 아까와 달리 노드의 갯수가 Hello에서 5개 그리고 공백 1개, world 5개로 총 11개로 구성되는 대신 Hello와 world 2개로 축약이 되는 것을 볼 수 있습니다. 이 덕분에 Radix 트라이는 기존의 트라이 구조보다 훨씬 더 가벼운 자원 사용을 요함을 알 수 있습니다.

Radix Trie 이전에 선행되어야 하는 지식들

이후 조금 더 체계적인 구현을 위한 선행 지식들을 찾아보았습니다. 기본적으로 알고리즘의 작동 과정 자체를 수학적으로 나타내는 2가지 체계인 시간 혹은 공간 복잡도에 대한 개념이 필요하다. 그 외에도 프로그래밍 언어 관련한 지식, 특히 더욱 안전하게 개발하기 위해 OOP(객체 지향 프로그래밍) 등의 개념이 어느정도 잡혀있는 것이 좋다고 생각되었고 여기에서 저에게 부족한 점은 어떻게 데이터를 다루는지였습니다.

Radix Trie 구현

실제로 간단하게 Radix Trie를 구현을 해보았는데 가장 난관으로 보여졌던 부분은 저에게 있어서 노드와 트라이 하나하나의 데이터 구조를 어떻게 짜느냐였습니다. 기본적으로 객체 지향 프로그래밍(OOP)의 개념은 '그 자체를 기능적으로 나타내는 것'이라고 생각합니다. 하지만 루트 노드 등등 아직은 선행 지식이 부족함을 느꼈습니다. 오른쪽은 간단히 평소에 하던 JavaScript로 구현한 각각의 클래스 구조입니다. 각각 새로운 단어를 처리하는 register 함수와 그리고 나중에 새로운 문장이 입력되었을 때 사용하는 search 함수를 포함하고 있습니다.



스스로 프로그래밍한 알고리즘과 알려져 있는 알고리즘 비교

항상 스스로 먼저 해보는 것이 저의 실질적인 실력 향상에 도움을 준다고 믿고 있기 때문에 아무것도 보지 않고 순전히 개인의 실력으로만 프로그래밍한 결과 위키백과 등 온라인에 나와있는 알고리즘의 예제 구현과는 꽤나 다른 구조를 가지고 있었습니다. 급한 구현도 맞지만 개인적으로 어떤 부분에서 차이가 있었는지 알아보았습니다.

구분	스스로 프로그래밍한 알고리즘	공개적으로 알려진 구현
문장의 단어 반복	For 문 사용, 반복의 길이를 알 수 있음	While 문 사용, 반복의 길이를 알 수 없음
사용한 프로그래밍 이론	OOP (객체 지향 프로그래밍)	단순 트라이
메모리 사용 (적을수록 좋음)	많음	적음
구현 복잡도 (낮을수록 좋음)	비교적 높음	비교적 낮음
구현 성공률	예외 처리 불가	예외 처리 불가

```

// Left pane: JSON structure of the trie
{
  "vel": {
    "tel": {
      "vel": {
        "loren": {
          "aliquam": {
            "tristique": {
              "at": {
                "a": {
                  "nulla": {}
                }
              }
            }
          }
        }
      }
    }
  }
}

// Right pane: JavaScript code for RadixTrie
module.exports = class RadixTrie {
  tree = {}

  tokenize(sentence) {
    const words = sentence
      .trim()
      .split(' ')
      .map(word => word.replace(/[^\w.]/mg, ''))
      .toLowerCase()
    return words
  }

  register(sentence) {
    const tokens = this.tokenize(sentence)
    let root = this.tree

    for (let i = 0, l = tokens.length; i < l; i++) {
      const token = tokens[i]

      if (!root[token]) {
        root[token] = {}
      }

      root = root[token]
    }
  }

  search(sentence) {
    const tokens = this.tokenize(sentence)
    let root = this.tree

    for (let i = 0, l = tokens.length; i < l; i++) {
      const token = tokens[i]

      if (!root[token]) {
        root = root[token]
      } else {
        return 0
      }
    }

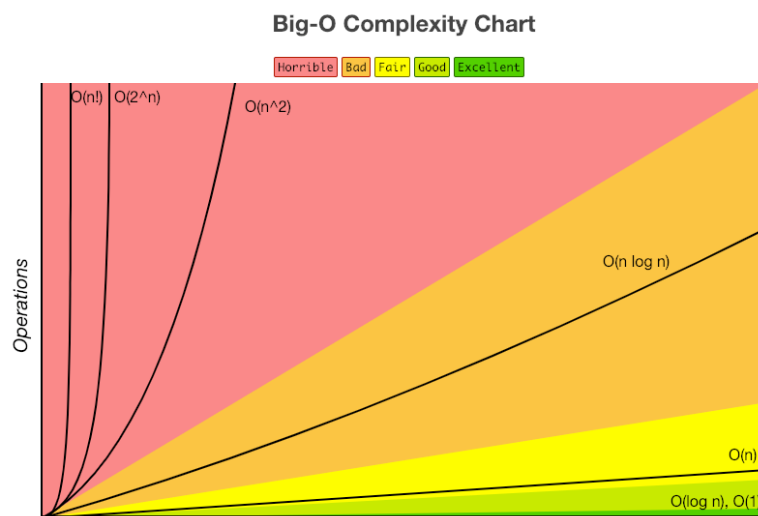
    if (root) {
      return 1
    }
  }
}

```

위는 실제로 간단히 직접 구현해본 Radix Trie의 일부 소스코드와 실제 결과값입니다. 기존에 알고리즘을 실제로 배우지 않았지만 어느정도 루트 노드까지는 구현이 오랜 시간없이 끝나게 되었습니다. 하지만 여전히 복잡한 소스코드와 비교적으로 높은 메모리 사용률에서 짐작할 듯 있듯이 아직 나아갈 점이 많이 보입니다.

작성한 알고리즘 및 메모리 사용률 등 성능 분석

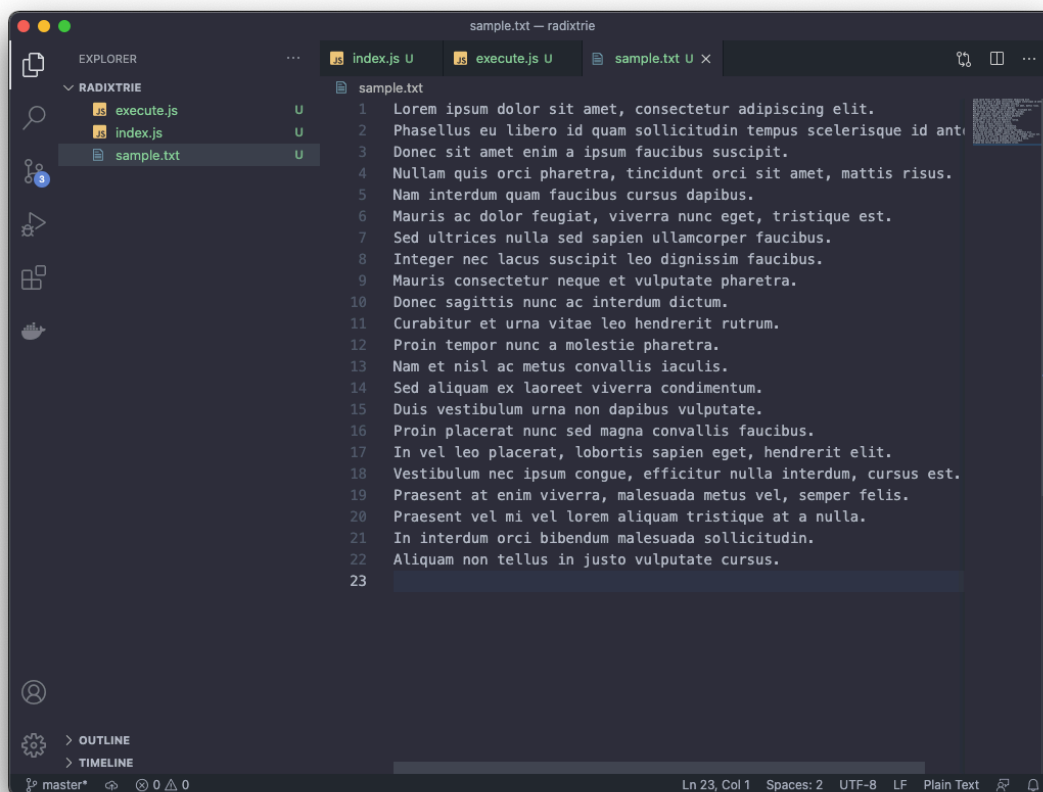
알고리즘 비교를 위해 몇 가지 요소를 살펴보고 위의 표를 완성하기까지 온라인에서 많은 사람들이 보는 평가 요소들을 찾아보았습니다. 먼저 위에서 말한 시간 복잡도와 공간 복잡도가 바로 제가 처음으로 판단할 요소입니다. 먼저 시간 복잡도는 문제 해결을 위해 걸리는 시간 그리고 얼마나 많은 함수가 알고리즘 실행 중에 성장을 하였는지 나타나는 지표로 판단할 수 있다. 함수의 성장이란 여기에서 얼마나 많은 함수가 동시다발적으로 실행되었는가를 나타냅니다.



Radix Trie는 텍스트 검색 알고리즘에서 전체 텍스트를 분류하지 않고 단어 마디마다 카테고리화를 시켜 메모리와 시간을 최적화시킵니다. 그렇기 때문에 꼭 전체 문장을 일일이 비교하지 않고 단어 단위로 같은 것을 찾아가기 때문에 훨씬 빠른 시간 내에 텍스트 검색을 완료할 수 있게 됩니다. 저와 같은 경우에는 알고리즘의 코드 중간에 입력값이 올바른지 확인하는 로직을 포함했기 때문에 검증 함수를 한 번 더 통과해야 하므로 정확히 2배의 Big-O 지표로 표현할 수 있습니다. 즉, 다음과 같이 나타낼 수 있습니다: $O(2n^2)$

그 외에도 구현 방법 상에서 객체 지향 프로그래밍 등의 요소가 있으나 알고리즘에 직접적으로 연관된 것이 아닌 프로그래밍 방법론 중 하나이기 때문에 자세히는 다루지 않습니다. 객체 지향 프로그래밍을 사용하면서 코드 자체의 제약을 많이 걸어 안정성을 확보할 수 있었지만 동시에 수많은 객체가 생성되기에 메모리 사용률 또한 기본적인 큐나 스택 구조보다 커지게 됩니다.

Radix Trie로 텍스트를 정렬시키기 위한 컴퓨터에서의 텍스트 저장 및 표현 형태 그리고 검색 알고리즘 분석



이제 보통의 텍스트 검색을 넘어 정렬을 효율적으로 구현하기 위해서 먼저 텍스트가 컴퓨터에서 어떻게 표현되는지 알아보도록 하겠습니다. 텍스트는 컴퓨터에서 단순히 숫자에 불과하기 때문에 이 특징을 이용하는 경우에 단순히 조건문을 사용하는 것보다 훨씬 더 빠른 검색 알고리즘을 구성할 수도 있을 것입니다. 또한 추가적인 성능 향상을 위해 프로그래밍 언어가 어떻게 작동하는지를 익혀야 합니다. 온라인 검색을 통해 대다수의 사람들이 프로그래밍 언어 내부에서 '만약 텍스트가 지정한 텍스트로 시작하는지' 확인하는 `startsWith` 함수보다 `slice` 등 기본적인 문자열 함수가 훨씬 빠르다는 것을 입증하였습니다. 다음은 그 결과입니다:

<code>indexOf</code>	113,625 ops/sec ±3.51% (78 runs sampled)
<code>startsWith</code>	22,522,847 ops/sec ±0.20% (94 runs sampled)
<code>lastIndexOf</code>	19,520,195 ops/sec ±0.41% (92 runs sampled)
<code>substring</code>	1,368,227,502 ops/sec ±0.07% (95 runs sampled)
<code>slice</code>	1,360,307,286 ops/sec ±0.07% (93 runs sampled)
<code>regex</code>	26,319,865 ops/sec ±1.17% (91 runs sampled)
<code>comp regex</code>	19,868,632 ops/sec ±3.01% (87 runs sampled)

Fastest is substring

그 외에도 텍스트는 바이너리와 txt 파일(일반 텍스트 파일)에서 저장되는 형식은 어느정도 제각각입니다. 하지만 가장 기본적인 단계에서 모두 그저 0과 1로 이루어진 비트이며 여기에서 더 빠른 포맷을 찾더라도 크기에 따라 변환을 하는데에 소비되는 시간이 더 길다고 생각되어 따로 다루지는 않았습니다. 또한 최근에는 HTTP 등 웹의 발전으로 대부분 텍스트는 8비트를 기준으로 저장된다고 알려져 있습니다. 따라서 결론적으로 텍스트를 다루기 위해 크게 고려해야 할 사항으로 프로그래밍 언어의 문자열 함수 사용을 고려하게 되었습니다.

Radix Trie를 통한 텍스트 정렬

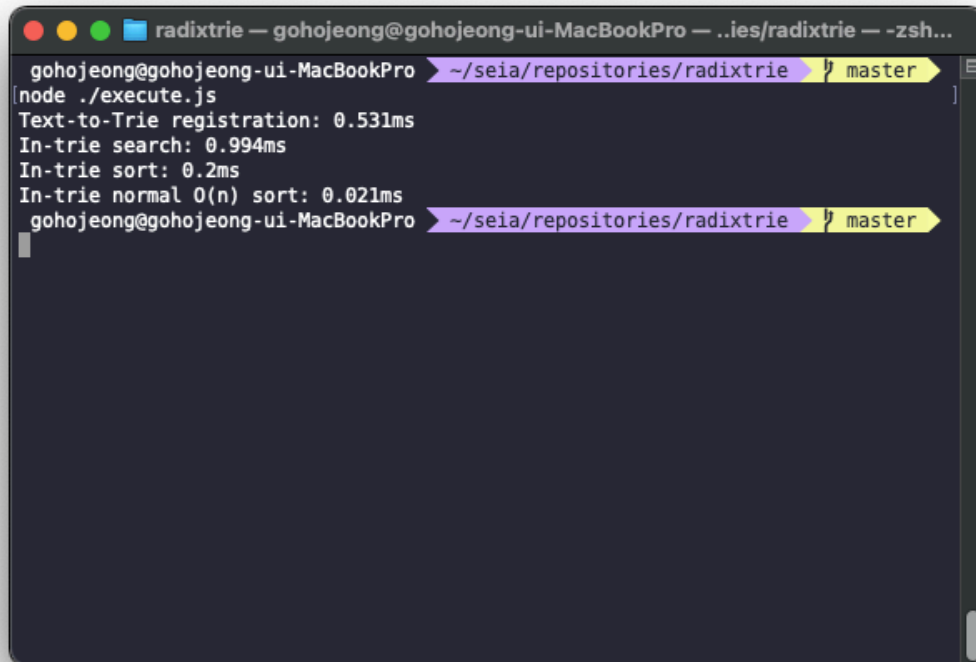
Radix Trie를 통한 텍스트 정렬을 위해 먼저 Lorem Ipsum 문장(의미가 없는 예시 문장)을 여러가지 생성했습니다. 모든 문장은 특이한 특수문자를 포함하고 있지 않으며 영문으로 이루어져 있습니다. (혹여나 포함하고 있더라도 자동으로 삭제 처리될 것입니다) 만일 한글 처리가 필요하다면 영문처럼 단순한 띄어쓰기가 아닌 더 복잡한 기준이 필요하게 될 것입니다.

정렬은 일반적인 문장 정렬과 다르게 Trie 자료형의 특성상 매순간마다 처음부터 필요한 연산 과정이 절약되기 때문에 더 빠르다고 생각합니다. 즉, 그 특성을 사용하면 하나의 복잡한 배열을 만들고 1차원으로 만들거나 혹은 처음부터 순차적으로만 처리하는 방법이 있습니다. 두 알고리즘 모두 공통적으로 문장 단위가 아닌 단어 단위로 여러번 판단한다는 특징을 가지고 있기 때문에 다음의 그림으로 설명이 됩니다. 오른쪽과 같은 그림에서 1차 노드라고 할 수 있는 Apple과 Banana만 비교하면 되는 것입니다. 그렇기 때문에 전체를 비교하는 기존의 문장 정렬과 달리 시간적인 부분에서 이익이 있을 것이라고 생각합니다.



반대로 단점이라면 자료형 특성상 모든 부분이 따로따로 되어 있기 때문에 모든 부분을 추적하여 다시 원래의 문장으로 만드는 것이 단점이 됩니다. 이는 DFS(깊이 우선 탐색) 기법을 사용하여 어느정도 완화할 수 있습니다. 여기에서 DFS는 위 예제에서 Apple 다음 Banana 순이 아닌 Apple 하위의 모든 노드를 우선적으로 탐색하여 가장 깊은 노드부터 순차적으로 탐색하는 기법입니다.

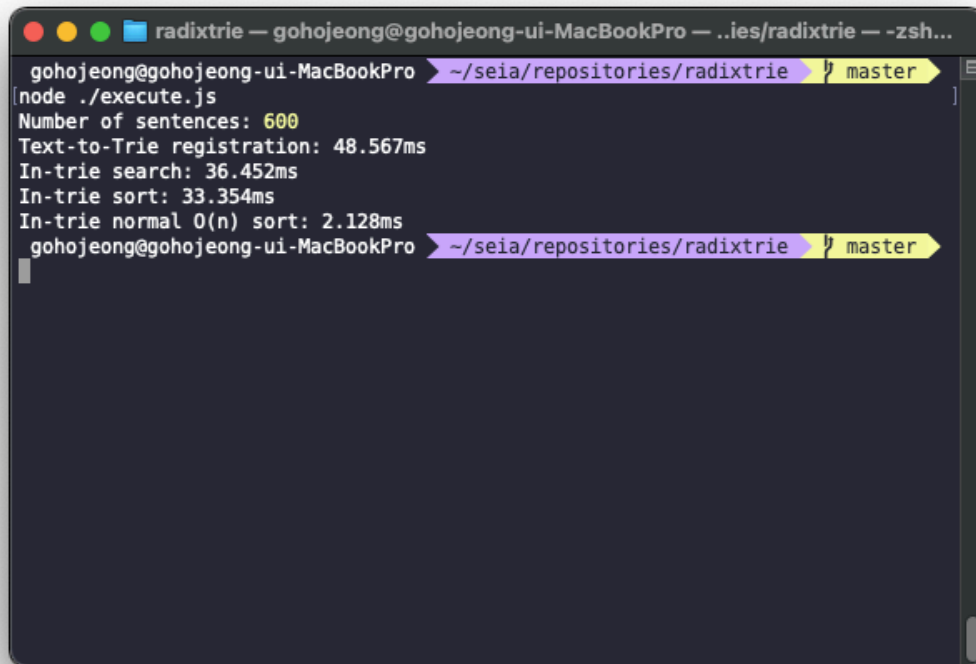
Radix Trie 텍스트 정렬 알고리즘 평가



```
gohojeong@gochojeong-ui-MacBookPro ~/seia/repositories/radixtrie master
node ./execute.js
Text-to-Trie registration: 0.531ms
In-trie search: 0.994ms
In-trie sort: 0.2ms
In-trie normal O(n) sort: 0.021ms
gohojeong@gochojeong-ui-MacBookPro ~/seia/repositories/radixtrie master
```

결과는 사실 예상보다 훨씬 더 좋지 않았습니다. 위의 Big-O 형식으로 표현해도 Radix Trie가 일반적인 반복문을 통하여 정렬을 시도하는 $O(n)$ 보다 훨씬 더 많은 반복이 필요했고 결과적으로 수많은 고차함수가 생성되어 복잡도 또한 높게 올라갔음을 볼 수 있습니다. Radix Trie를 정렬에 사용하는 예는 거의 없었습니다. 없는데에는 이유가 있지만 실제로 이렇게나 차이가 날 줄은 몰랐습니다. 자체적인 정렬 알고리즘 구현으로 0.2ms가 걸린 것이 $O(n)$ 으로는 0.021ms 밖에 걸리지 않았기 때문입니다. 하지만 여기에서 한 가지 의문점을 잡을 수 있습니다. Radix Trie는 비교할 가짓수가 늘어날 수록 효과적으로 가지치기를 하듯 이미 데이터가 어느정도 형태를 갖추고 있는 상태이기 때문에 더 빠르게 비교를 할 수 있다고 생각했습니다. 기존의 22개가 아닌 훨씬 더 많은 수의 문장이 있다면 어떨까 궁금했습니다.

결론 그리고 추가 실험 결과



```
gohojeong@gochojeong-ui-MacBookPro ~/seia/repositories/radixtrie master
node ./execute.js
Number of sentences: 600
Text-to-Trie registration: 48.567ms
In-trie search: 36.452ms
In-trie sort: 33.354ms
In-trie normal O(n) sort: 2.128ms
gohojeong@gochojeong-ui-MacBookPro ~/seia/repositories/radixtrie master
```

추가 실험을 통해 결론을 도출해낼 수 있었습니다. Radix Trie는 단순한 $O(n)$ 보다 훨씬 많은 반복문을 돌아야 합니다. 동시에 현재 알고리즘이 최적화되지 않았기 때문에 모든 데이터를 그 때 그 때마다 판별해야 하는 현재의 구현 방법에도 문제가 있다고 느껴졌습니다. 또한 이를 통해 각각의 알고리즘이 어느 부분에 효율적으로 작용할 수 있는지 배우고 $O(n)$ 표기법을 더욱 심화된 상태로 배울 수 있게 되었습니다. 비록 세운 가설은 실패하였지만 알고리즘의 용도를 알게 되었으니 추후 저의 타 프로젝트에서도 이 경험을 바탕으로 더 나은 프로그램을 설계할 수 있다고 생각합니다.

참조 및 참고

이 문서에서 사용된 단어나 출처 등을 정리하였습니다.

- Radix Trie는 기수 트라이라고도 합니다.
- 현재 문서에서 사용된 모든 도식은 draw.io에서 가져온 것입니다.

- 코드가 실행되는 환경은 macOS 11.0 Darwin kernel에서 컴파일된 aarch64 M1용 Node.JS 14.16.0 버전입니다.

기타 출처

- Fastify의 find-my-way, <https://www.npmjs.com/package/find-my-way>
- 위키백과의 트라이 (컴퓨팅) 문서, [https://ko.wikipedia.org/wiki/%ED%8A%B8%EB%9D%BC%EC%9D%B4_\(%EC%BB%B4%ED%93%A8%ED%8C%85\)](https://ko.wikipedia.org/wiki/%ED%8A%B8%EB%9D%BC%EC%9D%B4_(%EC%BB%B4%ED%93%A8%ED%8C%85))
- 영문 위키백과의 트라이 문서, https://en.wikipedia.org/wiki/Radix_tree
- 알고리즘의 시간 복잡도와 Big-O 쉽게 이해하기, <https://blog.chulgil.me/algorithm/>
- 위키백과의 시간 복잡도 문서, https://ko.wikipedia.org/wiki/시간_복잡도
- 빅오 표기법 (big-o notation) 이란 - 인생의 로그캣 티스토리 블로그, <https://noahlogs.tistory.com/27>
- GitHub Gists의 benchmark of String.startsWith equivalents in Node.JS 벤치마크 결과, <https://gist.github.com/dai-shi/4950506>
- Stackoverflow의 difference between text file and binary file 문서, <https://stackoverflow.com/questions/6039050/difference-between-text-file-and-binary-file>
- 칸 아카데미의 Storing text in binary, <https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:digital-information/xcae6f4a7ff015e7d:storing-text-in-binary/a/storing-text-in-binary>
- 영문 위키백과의 텍스트 문서, https://en.wikipedia.org/wiki/Text_file
- Stackoverflow의 Text vs Binary, <https://stackoverflow.com/questions/16670565/file-binary-vs-text>
- Lorem Ipsum Generator, <https://www.lipsum.com/feed/html>
- Carbon(코드 표현), <https://carbon.now.sh>