2008-2 고급계산이론 중간고사 답안

2008.11.09 김진일

1. Dynamic Table-Insert 문제

(아이디어)

Table에 차례로 entry를 차례로 채워 나가다가 table이 꽉 차면 크기가 1/c 배인 새로운 table을 할당하고 기존 entry들을 모두 복사한 후 이전 table을 버리고 새 table에 다음 entry를 추가한다.

(Pseudo code)

TableInsert(T, x)

```
1       if size[T] = 0
2       then    allocate table[T] with 1 slot
3               size[T] := 1
4       if num[T] = size[T]
5       then    allocate new-table with ⌊(1/c) · size[T]⌋
6               insert all items in table[T] into new-table
7               free table[T]
8               table[T] := new-table
9               size[T] := ⌊(1/c) · size[T]⌋
10      insert x into table[T]
11      num[T] := num[T]+1
```

(Remark)

위 pseudo code를 그대로 사용하는 경우 $\left\lfloor \frac{1}{c} \right\rfloor = 1$인 경우 (즉 $\frac{1}{2} < c < 1$) table의 크기가 $\frac{c}{1-c}$ 이상이어야 table을 새로 할당할 때 원래 테이블의 크기보다 1 이상 큰 table이 할당되어 새로운 entry를 추가할 수 있다. (이 경우에도 초반에는 load factor가 c 이하가 됨)

Table 확장시 1 이상 커지는 크기 유도:

$$\left\lfloor \frac{size}{c} \right\rfloor \geq size + 1$$
$$size+1 \leq \frac{size}{c}$$
$$c \cdot size + c \leq size$$
$$c \leq (1-c) \cdot size$$
$$size \geq \frac{c}{1-c}$$

예를 들어 c=0.9999 인 경우 table이 꽉 찼을 때 table의 크기가 $\frac{c}{1-c} = 9999$ 이상이어야 확장된 table의 크기가 기존 table의 크기에 비해 1 이상 증가하게 된다.

그러므로 table의 크기가 $\frac{c}{1-c}$ 미만인 경우 새 table의 크기를 기존 테이블의 크기보다 1 크게 할당하고 바로 다음 entry를 추가하면 table이 꽉 차게 되면서 조건을 만족시킬 수 있다.

(load factor)

수학적 귀납법 사용. 처음 entry 추가 후 load factor=1이므로 c보다 크다. i-1번째 entry 추가 후 load factor가 c 이상이라고 가정하자. 이전 table에 빈 공간이 남아있었던 경우 i번째 entry를 빈 공간에 넣게 되고 이 경우 load factor는 여전히 c 이상이다. Table이 확장된 경우 table의 크기는 1/c 이하로 증가하게 되므로 이 경우에도 load factor는 c 이하가 된다.

(Amortized cost)

분석을 위해 편의상 1/c가 정수라고 가정하자. i번째 연산의 수행 시간은 i가 (1/c)의 지수승에 해당할 때 table이 확장되면서 i만큼의 시간이 걸리고 그렇지 않은 경우 1만큼의 시간이 걸린다. 그러므로 n번의 Table-Insert 연산의 총 연산량은

$$\sum_{i=1}^{n} ci \le n + \sum_{j=0}^{\left\lfloor \log_{(\frac{1}{c})} n \right\rfloor} (\frac{1}{c})^j < n + (\frac{1}{c})^{\left\lfloor \log_{(\frac{1}{c})} n \right\rfloor + 1} < n + \left(\frac{1}{c}\right) n = \left(1 + \frac{1}{c}\right) n$$

그러므로 각 연산의 amotized cost는 (1+1/c)로 O(1)이 되고 n번 Table-Insert의 최악의 수행시간은 O(n)이 된다.

2. Baker-Bird 알고리즘

(알고리즘 설명)

(1) Row matching

P의 각 행을 패턴으로 하는 패턴 집합에 대하여 T의 각각의 행을 읽으면서 Aho-Corasick 알고리즘을 적용.

(P의 각 행 p_i에 해당하는 string에 id $r(p_i)$를 부여하고 T[i, j-m+1..j]가 P의 k번째 행인 경우 $R[i, j]=r(p_k)$로 할당)

(2) Column matching

앞에서 계산한 R의 각 열에 대하여 KMP 알고리즘을 적용하여 $P'=r(p_1)r(p_2)...r(p_m)$를 찾음 P'가 나타나는 위치를 기준으로 T에서 P가 나타남을 알 수 있다.

(시간복잡도)

(1) Row matching

P의 각 행에 대하여 Aho-Corasick 자료구조 생성에 $O(|\sum| m^2)$ 시간

T의 각 행에 대해 AC 적용하는데 $O(n)$가 걸리므로 총 $O(n^2)$ 시간

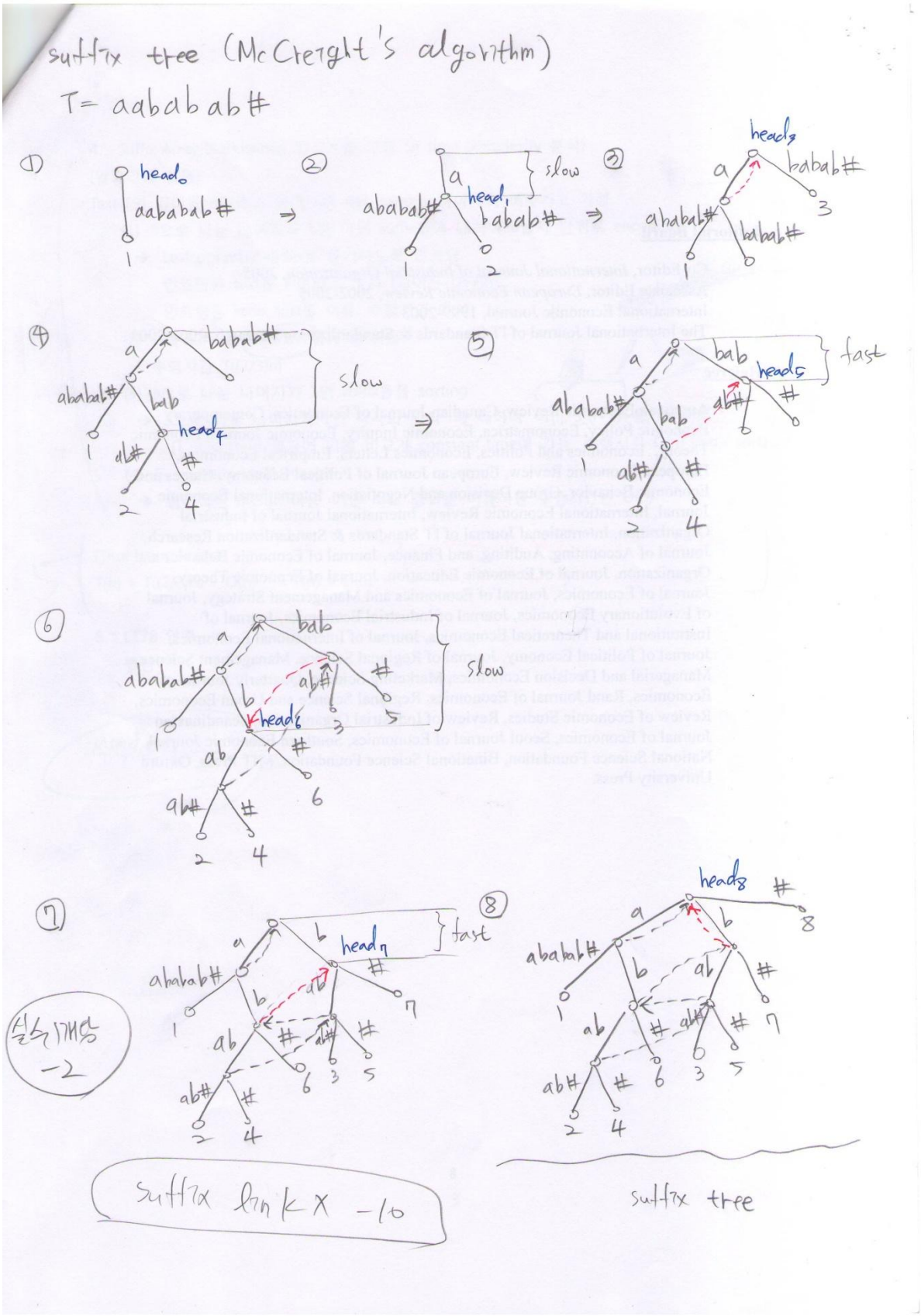
(2) Column matching

KMP preprocessing O(m)

각 열에 대해 $O(n)$ 시간이 걸리므로 총 $O(n^2)$ 시간이 소요

총 수행 시간은 $O(|\sum| m^2 + n^2)$이 된다.

3. Suffix Tree (McCreight 알고리즘)



4. Suffix Array (Karkkainen 알고리즘 설명 및 time complexity 분석)

(알고리즘 설명)

Text T의 길이를 n이라고 하고 n은 0을 padding하여 3의 배수라고 가정.

(1) 3으로 나눈 나머지가 1이 아닌 suffix들에 대하여 3글자 단위로 encoding

➔ Lexicographic order로 증가하도록 인코딩.

인코딩된 text를 T'이라고 하자. (길이는 (2/3)n)

인코딩은 radix sort를 이용. 수행시간 O(n)

(2) T'의 suffix를 recursive하게 sorting

수행시간 T((2/3)n)

(3) 3으로 나눈 나머지가 1인 suffix들을 sorting

➔ 3으로 나눈 나머지가 1인 각각의 위치 i에 대하여 <T[i], S_(i+1)>를 sorting

(T'의 sorting 결과에 의해 비교가 constant time에 가능. 수행시간 O(n) (radix sort)

(4) 두 sorted array를 merge

➔ 일반적인 merge 이용. 기존 비교 결과를 이용하면 수행시간은 O(n)

(설명 필요)

(Time complexity)

T(n) = T((2/3)n) + O(n) → T(n) = O(n)

5. LZ78 압축

T = ababb aaaba baaaa bbab

압축 결과

패턴번호	압축 텍스트	출력
1	a	(0,a)
2	b	(0,b)
3	ab	(1,b)
4	ba	(2,a)
5	aa	(1,a)
6	bab	(4,b)
7	aaa	(5,a)
8	abb	(3,b)
	ab	(3, null)

6. Off-line minimum problem

(a) Find the returned keys

Input : 4, 8, E, 3, 2, E, 9, 6, E, E, 1, 7, E, 5

Return : 4 2 3 6 1

(b) Off-line algorithm

(아이디어) 1부터 n까지 1번씩 insert되므로 가장 작은 숫자부터 차례로 돌면서 extract되는 위치를 계산한다. 이때, i가 extract되는 위치는 i 이후에 호출된 extract 중 아직 리턴값이 결정되지 않은 가장 가까운 extract이다.

이를 빠르게 찾기 위해 각각의 연속된 insert의 원소를 묶어 집합으로 묶는다. (j번째 extract와 j+1번째 extract 사이의 원소들의 집합을 K_j라고 표기) j번째 extract의 리턴값이 결정되면 extract 앞뒤의 집합을 합친다. (앞의 집합은 제거하고 합집합은 뒤의 집합으로 할당) 각 K_j를 union disjoint set을 이용하여 구현하면 임의의 i에 대하여 i가 extract될 위치는 i가 포함된 집합이 K_p라면 p번째 extract가 된다.

(Pseudo code) (CLRS p.519)

```
Off-Line-Minimum (m, n)
1       for i=1 to n
2           do determine j such that j ∈ K_j
3              if j ≠ m+1
4                 then extracted[j] :=i
5                      Let l be the smallest value greater than j for which set K_l exists
6                      K_j := K_j ∪ K_l
7       return extracted
```

(Correctness)

i에 대하여 extract 위치를 찾을 때 이미 리턴값이 결정된 extract들은 모두 i보다 작은 값들이 리턴값이다. 그러므로 i가 리턴될 위치는 i가 입력된 이후에 리턴값이 결정되지 않은 첫 번째 extract이다. 이 때 i가 K_p에 포함되어 있다면 K_p는 i가 처음에 속했던 집합부터 리턴값이 결정되지 않은 extract가 나올 때까지의 집합의 합집합이다. 그러므로 p가 리턴값이 결정되지 않은 첫 번째 extract가 되므로 i에 대하여 정확한 extract 위치를 계산한다. 이를 반복하면 m개의 extract에 대해 모두 정확한 extract 위치를 계산할 수 있다.

(Time Complexity)

모든 n개의 원소를 집합으로 구성하는데 O(n) 시간이 걸리고 m번 find를 하는데 각각 걸리는 시간이 O(alpha(n))이므로 시간 복잡도는 O(n+m*alpha(n))이다. (alpha(n) 대신 lg*n 가능)