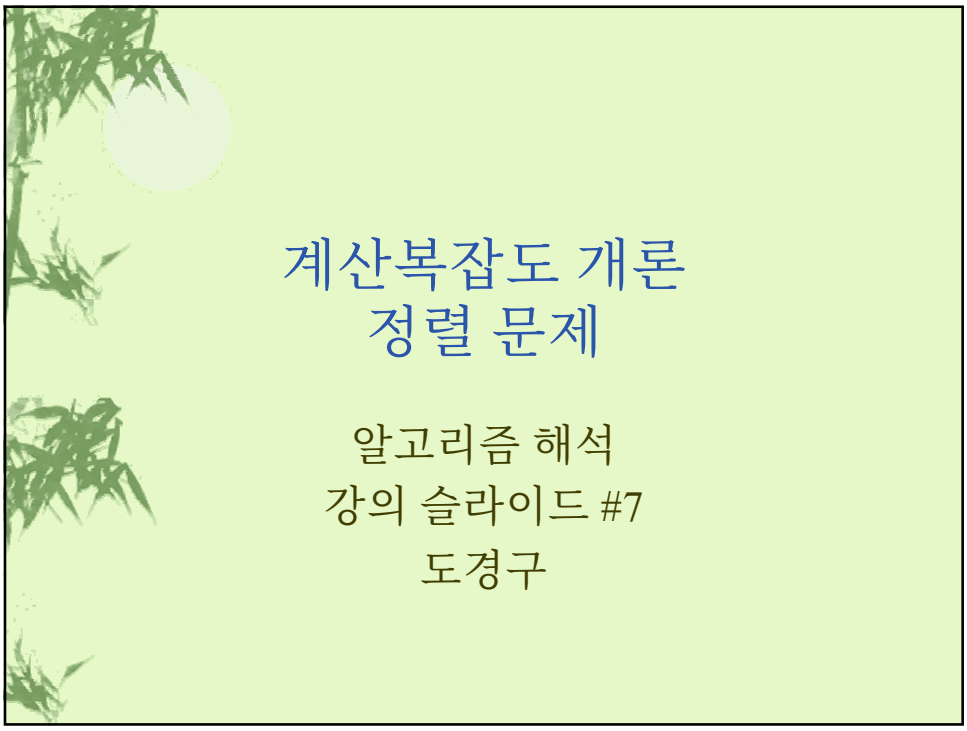


계산복잡도
Computational Complexity

- 알고리즘의 분석
  - 어떤 특정 알고리즘의 효율(efficiency)을 측정
  - 시간복잡도(time complexity)
  - 공간복잡도(space/memory complexity)
- 문제의 분석
  - 일반적으로 “계산복잡도 분석”이란 이를 지칭
  - 어떤 문제에 대해서 그 문제를 풀 수 있는 모든 알고리즘의 효율의 하한(lower-bound)을 결정한다.

보기: 행렬곱셈 문제

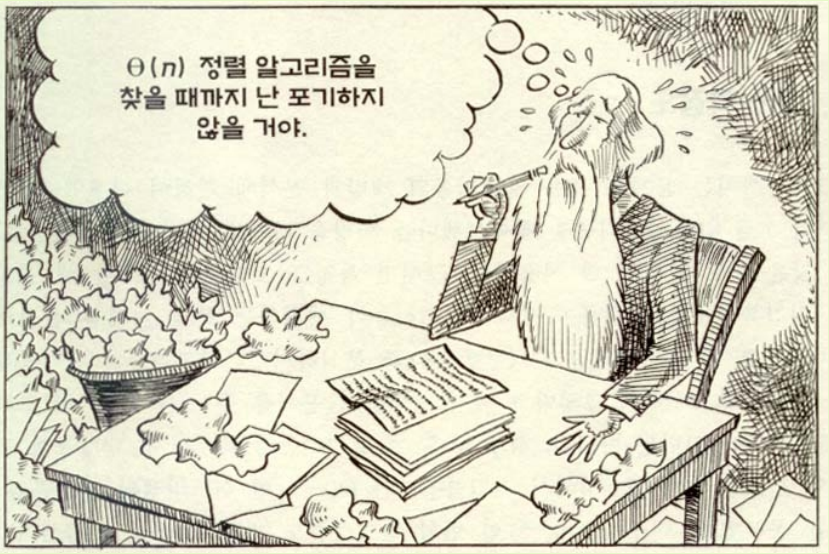
- 일반알고리즘: $\Theta(n^3)$
- Strassen의 알고리즘: $\Theta(n^{2.81})$
- Coppersmith/Winograd의 알고리즘: $\Theta(n^{2.38})$
- 이 문제의 복잡도의 하한은 $\Omega(n^2)$
- 더 빠른 알고리즘이 존재할까?
  - ✓ 아직 이 하한 만큼 좋은 알고리즘을 찾지 못하였고,
  - ✓ 그렇다고 하한이 이보다 더 큰지도 알아내지 못하였다.

2001-03-19 알고리즘 강의 슬라이드 7 3

계산복잡도

- 복잡도 하한이 $\Omega(f(n))$인 문제에 대해서 복잡도가 $\Theta(f(n))$인 알고리즘을 만들어 내는 것이 목표이다.
- 다시 말하면, 문제의 복잡도 하한보다 낮은 알고리즘을 만들어 낸다는 것은 불가능하다. (물론 상수적으로 알고리즘을 향상 시키는 것은 가능하다.)
- 보기: 정렬문제(sorting)
  - ✓ 교환정렬(Exchange sort): $\Theta(n^2)$
  - ✓ 합병정렬(Mergesort): $\Theta(n \lg n)$
  - ✓ 정렬문제의 시간복잡도 하한은 $\Omega(n \lg n)$ (키를 비교하여 정렬하는 경우에만 해당됨)
  - ✓ 이 정렬 문제의 경우는 하한 만큼의 시간 복잡도를 가진 알고리즘을 찾았다.

2001-03-19 알고리즘 강의 슬라이드 7 4



삽입정렬 알고리즘
Insertion Sort

- 이미 정렬된 배열에 항목을 끼워 넣음으로서 정렬하는 알고리즘
- 예) 8 4 2 7 9 5 13
- **알고리즘: 삽입정렬**
  - ✓ 문제: 비내림차순을 n개의 키를 정렬
  - ✓ 입력: 양의 정수 n; 키의 배열 S[1..n]
  - ✓ 출력: 비내림차순으로 정렬된 키의 배열 S[1..n]

도경구역, 알고리즘, 사이텍미디어, 1999

삽입정렬 알고리즘

```
void insertionsort(int n, keytype S[]){
  index i,j;
  keytype x;
  for(i=2; i<=n; i++){
    x = S[i];
    j = i - 1;
    while(j>0 && S[j]>x){
      S[j+1] = S[j];
      j--;
    }
    S[j+1] = x;
  }
}
```

2001-03-19 알고리즘 강의 슬라이드 7 7

삽입정렬 알고리즘의 분석

- **최악의 경우 시간복잡도 분석 - 비교하는 횟수를 기준:**
  i가 주어졌을 때, while-루프에서 최대한 i-1번의 비교가 이루어진다. 그러면 비교하는 총 횟수는 최대한
  $$W(n) = \sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}$$
- **평균의 경우 시간복잡도 분석 - 비교하는 횟수를 기준:**
  i가 주어졌을 때, x가 삽입될 수 있는 장소가 i개 있다.

삽입할 장소의 인덱스	i	$i-1$	$\cdots$	2	1
비교 횟수	1	2	$\cdots$	$i-1$	$i-1$

  x를 삽입하는데 필요한 비교 횟수는:
  $$1\frac{1}{i} + 2\frac{1}{i} + \cdots + (i-1)\frac{1}{i} + (i-1)\frac{1}{i} = \frac{1}{i}\sum_{k=1}^{i-1} k + \frac{i-1}{i} = \frac{(i-1)i}{2i} + \frac{i-1}{i} = \frac{i+1}{2} - \frac{1}{i}$$
  따라서 정렬하는데 필요한 평균 비교 횟수는:
  $$\sum_{i=2}^{n} \left(\frac{i+1}{2} - \frac{1}{i}\right) = \sum_{i=2}^{n} \frac{i+1}{2} - \sum_{i=2}^{n} \frac{1}{i} \approx \frac{(n+4)(n-1)}{4} - \ln n \approx \frac{n^2}{4}$$
- 공간복잡도 분석: 저장장소가 추가로 필요하지 않다. 따라서 M(n) = Θ(1). - 제자리정렬(in-place sorting) = 추가로 소요되는 저장장소의 크기가 상수인 경우.

2001-03-19 알고리즘 강의 슬라이드 7 8

선택정렬 알고리즘
Selection Sort

- 문제: 비내림차순으로 n개의 키를 정렬
- 입력: 양의 정수 n; 키의 배열 S[1..n]
- 출력: 비내림차순으로 정렬된 키의 배열 S[1..n]

```
void selectionsort(int n, keytype S[]){
  index i,j,smallest;
  for(i=1; i<=n-1; i++){
    smallest = i;
    for(j=i+1; j<=n; j++){
      if(S[j]<S[smallest])
        smallest = j;
    exchange S[i] and S[smallest];
  }
}
```

2001-03-19 알고리즘 강의 슬라이드 7 9

선택정렬 알고리즘의 분석

- **모든 경우 시간복잡도 분석 - 비교하는 횟수를 기준:** i가 1일 때 비교횟수는 n-1, i가 2일 때 비교횟수는 n-2,…, i가 n-1일 때 비교횟수는 1이 된다. 이를 모두 합하면, $\frac{n(n-1)}{2}$이다. 따라서

$$T(n) = \frac{n^2}{2}$$

- **모든 경우 시간복잡도 분석 - 지정(assignment)하는 횟수를 기준:** 1번 교환하는데 3번 지정하므로

$$T(n) = 3(n-1)$$

2001-03-19 알고리즘 강의 슬라이드 7 10

$\Theta(n^2)$ 정렬알고리즘의 비교

알고리즘	비교횟수	지정횟수	추가저장장소사용량
교환정렬	$T(n) = n^2/2$	$W(n) = 3n^2/2$ $A(n) = 3n^2/4$	제자리정렬
삽입정렬	$W(n) = n^2/2$ $A(n) = n^2/4$	$W(n) = n^2/2$ $A(n) = n^2/4$	제자리정렬
선택정렬	$T(n) = n^2/2$	$T(n) = 3n$	제자리정렬

- 삽입정렬은 교환정렬 보다는 항상 최소한 빠르게 수행된다고 할 수 있다.
- 선택정렬이 교환정렬 보다 빠른가? - 일반적으로는 선택정렬 알고리즘이 빠르다고 할 수 있다. 그러나 입력이 이미 정렬되어 있는 경우, 선택정렬은 지정이 이루어지지만 교환정렬은 지정이 이루어지지 않으므로 교환정렬이 빠르다.
- 선택정렬 알고리즘이 삽입정렬 알고리즘 보다 빠른가? - n의 크기가 크고, 키의 크기가 큰 자료구조 일 때는 지정하는 시간이 많이 걸리므로 선택정렬 알고리즘이 더 빠르다.

2001-03-19 알고리즘 강의 슬라이드 7 11

한번 비교하는데 최대한 하나의 역을 제거하는 알고리즘의 하한선

- n개의 별개의(다른) 키를 정렬하는데, 정렬할 키들을 단순히 양의 정수 1, 2,…, n이라고 가정해도 보편성이 희생되지 않는다.
- n개의 양수는 $n!$개의 순열(permutation)이 존재한다. 다시 말하면, $n!$가지의 다른 순서로 배열할 수 있다는 뜻이다.
- k_i를 i번째 자리에 위치한 정수라고 할 때, 순열은 $[k_1, k_2,\ldots, k_n]$으로 나타낼 수 있다. 예를 들면, [3, 1, 2]는 $k_1 = 3, k_2 = 1, k_3 = 2$로 표시할 수 있다.
- $i < j$와 $k_i < k_j$의 조건을 만족하는 쌍(pair) (k_i, k_j)를 순열에 존재하는 역(inversion)이라고 한다. 예를 들어 순열 [3, 2, 4, 1, 6, 5]에는 몇 개의 역이 존재할까? 답: 5개

2001-03-19 알고리즘 강의 슬라이드 7 12

한번 비교하는데 최대한 하나의 역을 제거하는 알고리즘의 하한선

- **정리 7.1**: 서로 다른 n개의 키를 키끼리 비교하여 정렬할 때, 매번 비교할 때 마다 기껏해야 하나의 역만을 제거하는 알고리즘은
  최악의 경우에 최소한 $\frac{n(n-1)}{2}$ 의 비교를 수행해야 하고
  평균적으로 최소한 $\frac{n(n-1)}{4}$ 의 비교를 수행해야 한다.

2001-03-19 알고리즘 강의 슬라이드 7 13

한번 비교하는데 최대한 하나의 역을 제거하는 알고리즘의 하한선

증명:

- **경우 1**: (최악의 경우)
  $n(n\text{-}1)/2$개의 역을 가진 순열이 존재한다고 보이면 충분하다. 왜냐하면 그러한 순열이 있으면 알고리즘이 그만큼의 역을 제거해야 하고, 따라서 최소한 그만큼의 비교를 수행해야 하기 때문이다. 그러한 순열은: $[n, n\text{-}1,\ldots, 2, 1]$.
- **경우 2**: (평균적으로)
  어떤 임의의 순열 $P = [k_1, k_2,\ldots, k_n]$이 있을 때, 그 순열을 뒤집어 놓으면 (transpose) $P^T = [k_n,\ldots, k_2, k_1]$이 된다. 이때 어떤 역이 되는 쌍(pair) (s, r)은 반드시 P에 속하거나, 아니면 P^T에 속하게 된다. 그러면 P와 P^T에는 정확하게 총 $n(n\text{-}1)/2$개의 역이 존재하게 된다. 따라서 P와 P^T에 존재하는 역의 평균 개수는 $n(n\text{-}1)/4$이 된다. 여기서 알고리즘이 매번 비교할 때 마다 기껏해야 하나의 역 만을 제거한다고 가정하였으므로, 알고리즘이 모든 역을 제거하기 위해서는 최소한 역의 개수 만큼의 비교를 평균적으로 수행해야 한다. 따라서 정렬하는데도 그 만큼의 비교를 수행해야 한다.

- 교환, 삽입, 선택 정렬은 한번 비교할 때 기껏해야 하나의 역 이상은 제거할 수 없으므로 시간복잡도가 최악의 경우 $n(n-1)$, 평균적으로는 $n(n-1)$ 보다 좋을 수 없다.

2001-03-19 알고리즘 강의 슬라이드 7 14

합병정렬 알고리즘 재검토

- 합병정렬(Mergesort)에서는 한번 비교해서 하나 이상의 역을 제거하는 경우가 있다. 예를 들어, 두 부분배열 [3,4]과 [1,2]를 합병해 보자. 3과 1을 비교한 후에 1은 배열의 첫번째 자리에 놓게 되고, 따라서 두개의 역인 (3,1)과 (4,1)을 동시에 제거하게 된다. 다음, 3과 2를 비교한 후에 2는 배열의 두 번째 자리에 놓게 되고, 따라서 두 개의 역인 (3,2)와 (4,2)를 동시에 제거하게 된다.
- 키를 비교하는 횟수를 기준으로 시간복잡도를 계산하면:

$$W(n) = A(n) = \Theta(n \lg n)$$

  키를 지정하는 횟수를 기준으로 (모든 경우) 시간복잡도를 계산하면: $T(n) \approx 2n \lg n$
- 공간복잡도: $M(n) = \Theta(n)$ - 제자리정렬이 아님.

2001-03-19 알고리즘 강의 슬라이드 7 15

빠른 정렬 알고리즘 재검토

- 키를 비교하는 횟수를 기준으로 시간복잡도를 계산하면:

$$W(n) = \Theta(n^2) \text{ and } A(n) \approx 1.38(n+1)\lg n$$

  키를 지정하는 횟수를 기준으로 (모든 경우) 시간복잡도를 계산하면: $A(n) \approx 3 \times 0.69(n+1)\lg n$
- 공간복잡도: 이 알고리즘은 합병정렬알고리즘과 비교해서 추가적인 배열이 필요하지 않는 장점이 있다. 그러나 합병정렬과는 달리 배열이 정확하게 반으로 나누어지지 않으므로, 한쪽의 부분배열을 정렬할 때, 다른 한쪽의 부분배열의 첫번과 마지막 인덱스를 활성레코드(activation record) 스택에 저장하여 둘 필요가 있다. 따라서 추가적으로 필요한 저장장소는 최악의 경우 $M(n) = \Theta(n)$ 가 된다. 따라서 제자리정렬 알고리즘이라고 할 수 없다. 그러나 추가적으로 필요한 저장장소가 $M(n) = \Theta(\lg n)$ 가 되도록 알고리즘 수정이 가능하다.

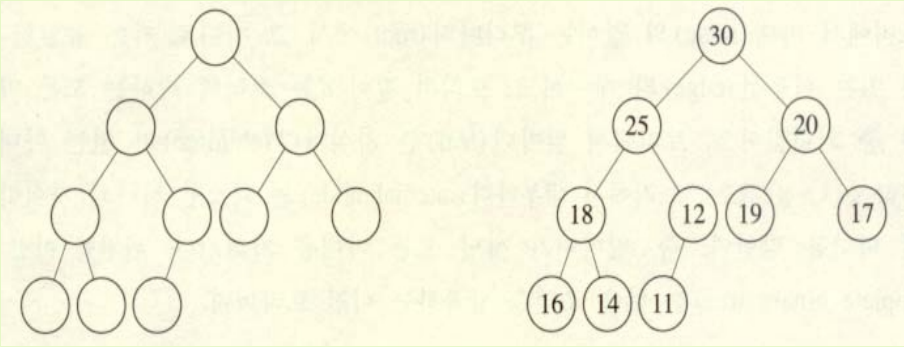
2001-03-19 알고리즘 강의 슬라이드 7 16

힙정렬(Heapsort) 알고리즘

- 완전이진트리(complete binary tree): 트리의 내부에 있는 모든 마디에 두 개씩 자식마디가 있는 이진 트리. 따라서 모든 잎의 깊이(depth) d는 동일하다.
- 실질적인 완전이진트리(essentially complete binary tree)
  - ✓ 깊이 d - 1까지는 완전이진트리이고,
  - ✓ 깊이 d의 마디는 왼쪽 끝에서부터 채워진 이진트리.
- 힙의 성질(heap property): 어떤 마디에 저장된 값은 그 마디의 자식마디에 저장된 값보다 크거나 같다.
- 힙(heap): 힙의 성질을 만족하는 실질적인 완전이진트리

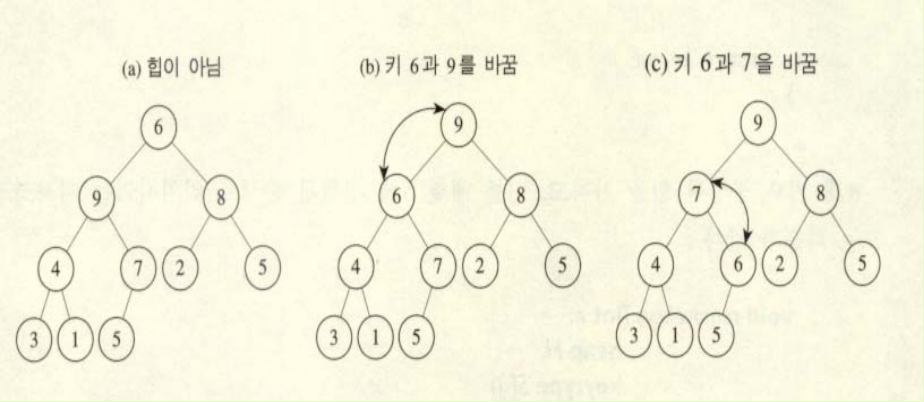
2001-03-19 알고리즘 강의 슬라이드 7 17

보기



2001-03-19 알고리즘 강의 슬라이드 7 18

보기: siftdown

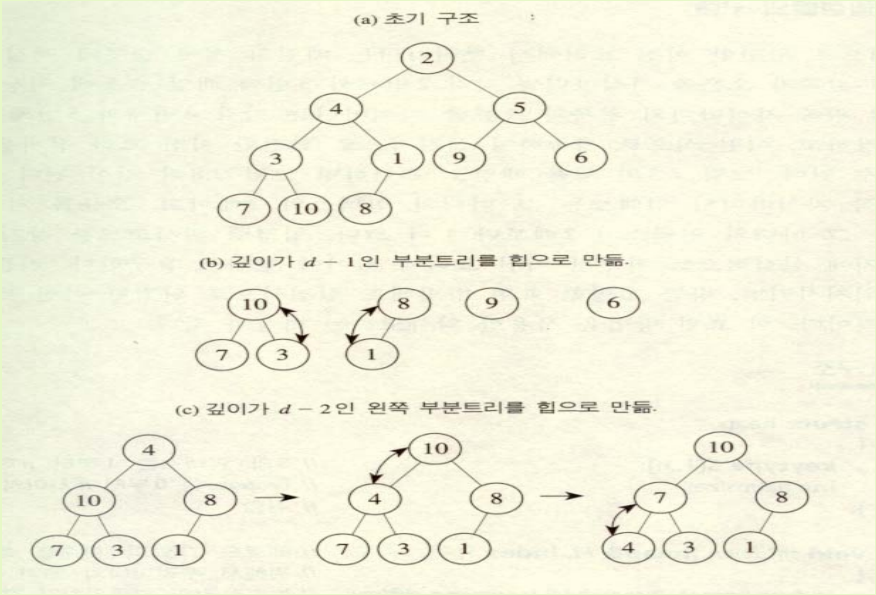


힙정렬

- **알고리즘: 힙정렬(Heapsort)**

```
void siftdown(heap& H){
  node parent, largerchild;
  parent = root of H;
  largerchild = parent's child containing larger key;
  while(key at parent is smaller than key at largerchild){
    exchange key at parent and key at largerchild;
    parent = largerchild;
    largerchild = parent's child containing larger key;
  }
}
keytype root(heap& H){
  keytype keyout;
  keyout = key at the root;
  move the key at the bottom node to the root;
  delete the bottom node;
  siftdown(H);
  return keyout;
}
```

보기: makeheap



힙정렬

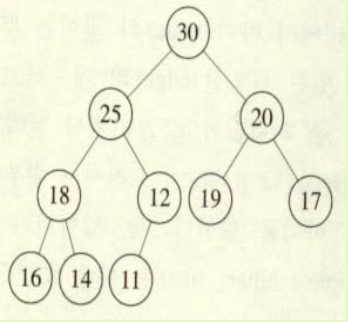
```
void removekeys(int n, heap H, keytype S[]){
  index i;
  for(i=n; i>=1; i--)
    S[i] = root(H);
}
void makeheap(int n, heap& H){
  index i;
  heap Hsub;
  for(i=d-1; i>=0; i--)
    for(all subtree Hsub whose roots have depth i)
      siftdown(Hsub);
}
void heapsort(int n, heap H, keytype S[]){
  makeheap(n,H);
  removekeys(n,H,S);
}
```

힙정렬 알고리즘의 공간복잡도

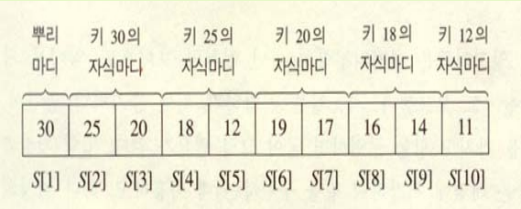
- 이 알고리즘이 제자리정렬 알고리즘인가? 다음과 같이 힙을 배열로 구현한 경우에는 제자리정렬 알고리즘이다.
  공간복잡도: $\Theta(1)$
- 실질적인 완전이진트리를 배열로 표현하는 방법:
  - ✓ 어떤 마디의 왼쪽 지식마디의 인덱스는 그 마디의 인덱스의 두 배이다.
  - ✓ 어떤 마디의 오른쪽 자식마디의 인덱스는 그 마디의 인덱스의 두 배보다 1만큼 크다.
- **알고리즘: 제자리**

```
void heapsort(int n, heap& H){
  makeheap(n,H);
  removekeys(n,H,H.S);
}
```


완전이진트리 ⇒ 배열



⇒



- **최악의 경우 시간복잡도 분석 - 비교하는 횟수를 기준:**
  - 단위연산: siftdown 프로시저에서의 키의 비교
  - 입력크기: n, 총 키의 개수
  - makeheap와 removekeys 모두 siftdown을 부르므로 따로 분석을 한다.
  - makeheap의 분석: $n = 2^k$라 가정.

    d를 실질적인 완전이진트리의 깊이라고 하면, $d = \lg n$. 이때 d의 깊이를 가진 마디는 정확히 하나이고 그 마디는 d개의 조상(ancestor)을 가진다. 일단 깊이가 d인 그 마디가 없다고 가정하고 키가 sift되는 상한값(upper bound)을 구해 보자.

depth	node수	키가 sift되는 최대횟수
0	2^0	$d-1$
1	2^1	$d-2$
2	s^2	$d-3$
⋮	⋮	⋮
j	2^j	$d-j-1$
⋮	⋮	⋮
$d-2$	2^{d-2}	1
$d-1$	2^{d-1}	0

2001-03-19 알고리즘 강의 슬라이드 7 25

따라서 키가 sift되는 총 횟수는 $\sum_{j=0}^{d-1} 2^j(d-j-1)$를 넘지 않는다. 이를 $\sum_{i=0}^{n} 2^i = 2^{n+1}-1$와 $\sum_{i=1}^{n} i2^i = (n-1)2^{n+1}+2$를 이용하여 계산해 보면,

$$\sum_{j=0}^{d-1} 2^j(d-j-1) = (d-1)\sum_{j=0}^{d-1} 2^j - \sum_{j=0}^{d-1} j2^j = 2^d - d - 1$$

여기에다 깊이가 d인 경우 sift될 횟수의 상한값인 d를 더하면, $2^d - 1 = n - 1$ 이 된다. 그런데 한번 sift될 때마다 2번씩 비교하므로 실제 비교횟수는 $2(n\text{-}1)$ 이 된다.

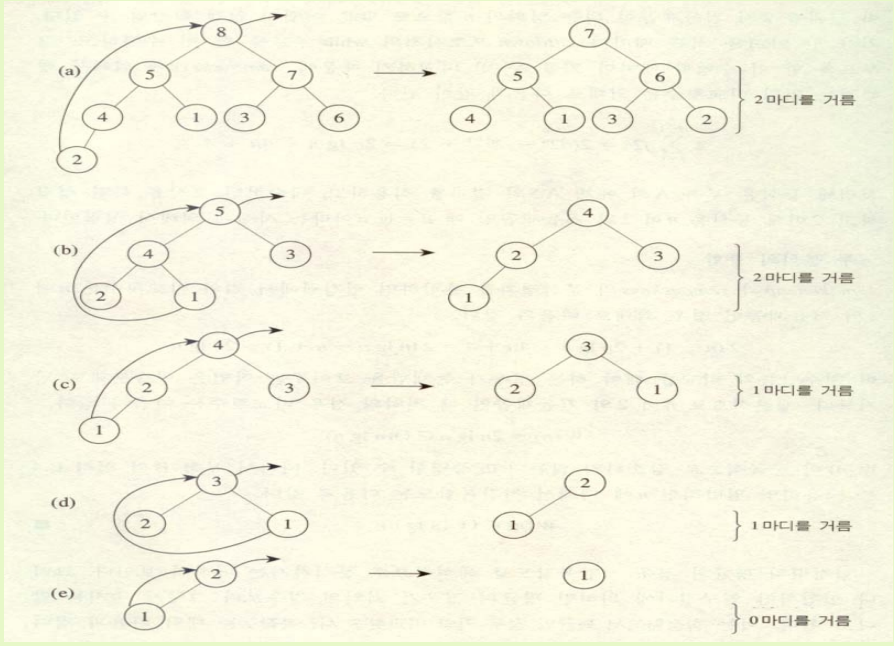
- removekeys의 분석: $n = 2^k$라 가정.

  먼저 $n = 8$이고 $d = \lg 8 = 3$인 경우를 생각해 보자. 처음 4개의 키를 제거하는데 sift되는 횟수가 2회, 다음 2개의 키를 제거하는데 sift되는 횟수가 1회, 그리고 마지막 2개의 키를 제거 하는 데는 sift되지 않았다. 따라서 총 sift횟수는 $1(2) + 2(4) = \sum_{j=1}^{3-1} j2^j$가 된다. 따라서 일반적인 경우는

$$\sum_{j=0}^{d-1} j2^j = (d-2)2^d + 2 = d\cdot 2^d - 2\cdot 2^d + 2 = n\lg n - 2n + 2$$

  가 된다. 그런데 한번 sift될 때 마다 2번씩 비교하므로 실제 비교횟수는 $2n\lg n - 4n + 4$이 된다.
- 두 분석의 통합: 키를 비교하는 총 횟수는 n이 2^k일 때 $2(n\text{-}1) + 2n\lg n - 4n + 4 = 2(n\lg n - 2n + 1) \approx 2n\lg n$ 을 넘지 않는다. 따라서 최악의 경우 $W(n) \in \Theta(2n\lg n)$. 일반적으로 모든 n에 대해서도 $W(n) \in \Theta(2n\lg n)$이라고 할 수 있는데, 왜냐하면 $W(n)$은 결국은 증가함수이기 때문이다.

2001-03-19 알고리즘 강의 슬라이드 7 26

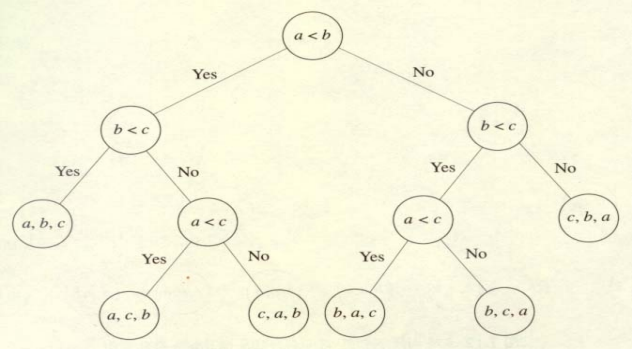


$\Theta(n \lg n)$ 알고리즘의 비교

알고리즘	비교횟수	지정횟수	추가저장장소사용량
합병정렬	$W(n) = A(n) = n \lg n$	$T(n) = 2n \lg n$	$\Theta(n)$
빠른정렬	$W(n) = n^2/2$ $A(n) = 1.38n \lg n$	$A(n) = 0.69n \lg n$	$\Theta(\lg n)$
힙정렬	$W(n) = A(n) = 2n \lg n$	$W(n) = A(n) = n \lg n$	제자리정렬

키의 비교만으로 정렬하는 경우 하한

- $n \lg n$ 보다 더 빠른 정렬 알고리즘을 개발할 수 있을까?
  - ✓ 키의 비교 횟수를 기준으로 하는 한, 더 빠른 알고리즘은 불가능하다.
- 정렬알고리즘에 대한 결정트리
  - ✓ 3개의 키를 정렬하는 알고리즘을 생각해 보자. 그 3개의 키를 각각 a,b,c 라고 한다면 다음과 같은 결정트리(decision tree)를 그릴 수 있다.

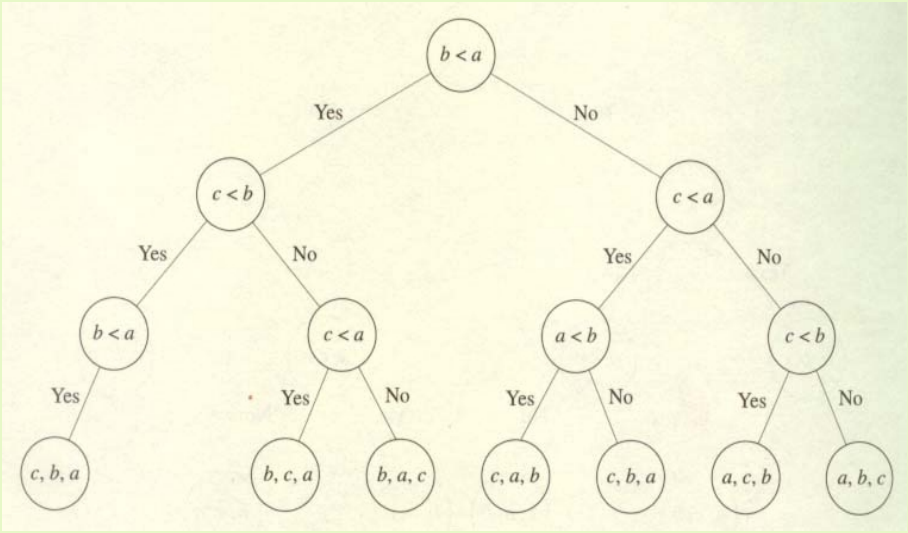


2001-03-19 알고리즘 강의 슬라이드 7 29

키의 비교만으로 정렬하는 경우 하한

- ✓ n개의 키를 정렬하기 위한 결정트리를 생각해 보자. 만약 n개의 키의 각 순열(permutation)에 대해서, 뿌리마디로부터 잎마디로 이르는 그 순열을 정렬하는 경로가 있는 경우, 그 결정트리는 유효하다(valid)고 한다. 즉, 크기가 n인 어떤 입력에 대해서도 정렬할 수 있다.
- ✓ 그러면 3개의 입력을 정렬하는 교환정렬 알고리즘의 결정트리는 어떤 모양을 가질까? 여기서 잘 살펴보면 교환정렬에서는 꼭 하지 않아도 될 비교를 하고 있다. 왜냐하면 교환정렬에서는 어떤 시점에서 비교가 이루어 질 때, 그 이전에 이루어졌던 비교의 결과를 전혀 알 수 없기 때문이다. 이러한 현상은 최적(optimal)이 아닌 알고리즘에서 주로 나타난다.
- ✓ 가지친 결정트리(pruned decision tree): 일관성 있는 순서로 결정을 내림으로서 뿌리마디로부터 모든 잎마디에 도달할 수 있는 경우. 다음에 있는 결정트리는 가지친(pruned) 결정트리이다.
- ✓ **보조정리 7.1**: n개의 서로 다른 키를 정렬하는 결정적(deterministic)알고리즘은, 그에 상응하는 정확하게 $n!$개의 잎마디를 가진, 유효하며 가지친 이진 결정트리가 존재한다.

2001-03-19 알고리즘 강의 슬라이드 7 30



최악의 경우 하한

- **보조정리 7.2**: 결정트리에 의해서 수행된 최악의 경우 비교횟수는 그 결정트리의 깊이와 같다.
- **보조정리 7.3**: 이진트리(binary tree)의 잎마디의 수가 m이고, 깊이가 d이면, $d \geq \lceil \lg m \rceil$이다.

  증명: d에 대하여 귀납법으로 증명. 우선 $2^d \geq m$임을 먼저 보인다.
  - 귀납출발점: $d = 0$: 마디의 수가 하나인 이진트리. 따라서 명백히 $2^d \geq 1$.
  - 귀납가정: 깊이가 d인 모든 이진트리에 대하서, $2^d \geq m$가 성립한다고 가정.
  - 귀납절차: 깊이가 $d+1$인 모든 이진트리에 대해서, $2^d \geq m'$임을 보이면 된다. 여기서 m'은 잎마디의 수이다.

    $2^{d+1} = 2 \times 2^d \geq 2m$ 귀납가정에 의해서 성립

    $\geq m'$ 각 부모다디는 기껏해야 자식마디 2개를 가지므로

    그러므로 $2^d \geq m$이 성립한다. 여기서 양변에 lg를 씌우면, $d \geq \lg m$이 된다. 그런데 d는 정수이므로, $d \geq \lceil \lg m \rceil$이 된다.

최악의 경우 하한

- **정리 7.2**: n개의 서로 다른 키를 비교함으로 만 정렬하는 결정적 알고리즘은 최악의 경우 최소한 $\lceil n \lg n - 1.45n \rceil$ 번의 비교를 수행한다.
  증명: 보조정리 7.1에 의하면, $n!$개의 잎마디를 가진 가지친, 유효한, 이진결정트리가 존재한다. 다시 보조정리 7.3에 의하면, 그 트리의 깊이는 ≥ $\lceil \lg (n!) \rceil$가 되고, 그러면

$$\begin{aligned} \lg(n!) &= \lg[n(n-1)(n-2)\cdots 2] \\ &= \sum_{i=2}^{n} \lg i \\ &\geq \int_{1}^{n} \lg x \, dx \\ &= \frac{1}{\ln 2}(n \ln n - n + 1) \\ &\geq n \lg n - 1.45n \end{aligned}$$

  따라서 보조정리 7.2에 의해서, 결정트리의 최악의 경우의 비교횟수는 그 트리의 깊이와 같다. 증명 끝.

정렬알고리즘의 분류

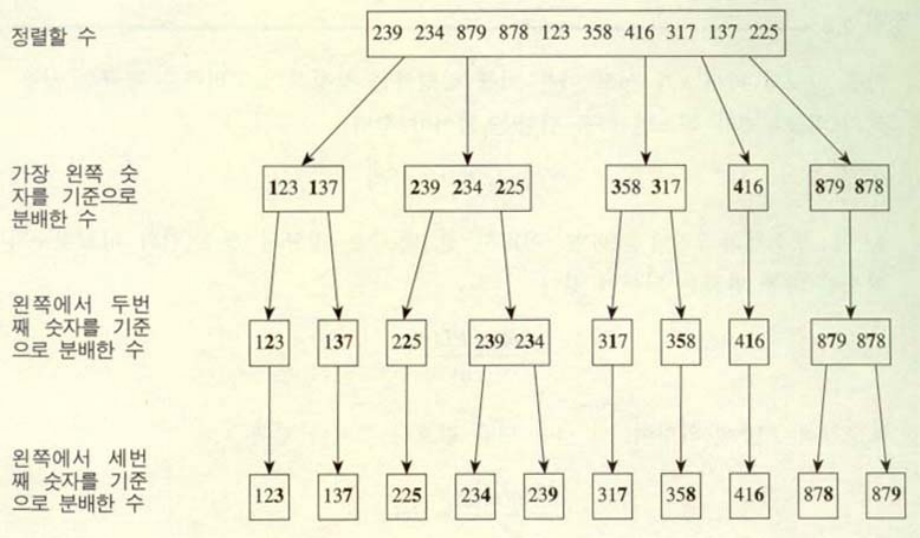
- **선택하여 정렬하는 부류의 알고리즘**
  - ✓ 순서대로 항목을 선택하여 적절한 곳에 위치시키는 알고리즘.
  - ✓ 교환정렬, 삽입정렬, 선택정렬, 힙정렬 알고리즘이 이 부류에 속한다.
  - ✓ 공간복잡도는 모두 $\Theta(1)$이다. 모두 제자리정렬 알고리즘
- **자리를 바꾸어서 정렬하는 부류의 알고리즘**
  - ✓ 인접해 있는 항목을 교환하여서(자리를 바꾸어서) 정렬이 수행되는 알고리즘.
  - ✓ 거품정렬(Bubble Sort($\Theta(n^2)$)), 합병정렬, 빠른정렬 알고리즘이 이 부류에 속한다.
  - ✓ 거품정렬을 제외하고는 모두 제자리정렬 알고리즘이 아니다.

분배에 의한 정렬: 기수정렬

- 키에 대해서 아무런 정보가 없는 경우에는 키들을 비교하는 것 이외에는 다른 방법이 없으므로 $\Theta(n \lg n)$보다 더 좋은 알고리즘을 만드는 것은 불가능하다.
- 그러나 키에 대한 어느 정도의 정보를 알고 있는 경우에는 상황이 조금 달라질 수 있다. 가령 키들이 음이 아닌 수이고 디지트(digit)의 개수가 모두 같다면, 첫번째 디지트가 같은 수끼리 따로 모으고, 그 중에서 두 번째 디지트가 같은 수끼리 따로 모으고, 마지막 디지트가지 이런 식으로 계속 모은다면, 각 디지트를 한번씩 만 조사를 하면 정렬을 완료할 수 있다. 다음에 예가 제시되어 있는데, 이런 방법으로 정렬하는 것을 "분배에 의한 정렬(sorting by distribution)" - 기수정렬(radix sort) - 이라고 한다.

2001-03-19 알고리즘 강의 슬라이드 7 35

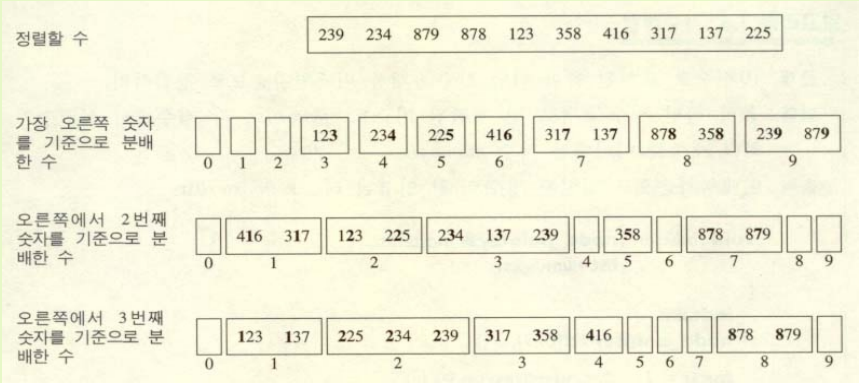
보기 - 기수정렬



2001-03-19 알고리즘 강의 슬라이드 7 36

기수정렬 알고리즘

- 이와 같이 정렬하는 경우 항상 뭉치(pile)를 구성하는 개수가 일정하지 않으므로 관리하기가 쉽지 않다. 이를 해결하기 위해서는 다음 예와 같이 끝에 있는 디지트부터 먼저 조사를 시작하면 된다.



2001-03-19 알고리즘 강의 슬라이드 7 37

기수정렬 알고리즘의 분석

- 단위연산: 뭉치에 수를 추가하는 연산
- 입력크기: 정렬하는 정수의 개수 = n, 각 정수를 이루는 디지트의 최대 개서 = $numdigits$
- 모든 경우 시간복잡도 분석:

$$T(n) = numdigits\ (n+10) \in \Theta(numdigits \times n)$$

따라서 $numdigits$가 n과 같으면, 시간복잡도는 $\Theta(n^2)$가 된다. 그러나 일반적으로 서로 다른 n개의 수가 있을 때 그것을 표현하는데 필요한 디지트의 수는 $\lg n$으로 볼 수 있다. 예: 주민등록번호는 13개의 디지트로 되어 있는데, 표현할 수 있는 개수는 10,000,000,000,000개 이다. 이 10조개의 번호를 기수정렬하는데 걸리는 시간은 $10{,}000{,}000{,}000{,}000 \times \log_{10} 10{,}000{,}000{,}000{,}000 =$ 130조

- 공간복잡도 분석
  - ✓ 추가적으로 필요한 공간은 키를 연결된 리스트로 만드는데 필요한 공간, 즉, $\Theta(n)$

2001-03-19 알고리즘 강의 슬라이드 7 38