Data structure [A05]

김종규, PhD

2017-04-03

Review

- Quiz problems

Outline

- Binary search and recursion
- Sorting algorithms
  - Selection sort
  - Mergesort
  - Heapsort
  - Quicksort

Complexity

- What is the complexity of the following algorithm (Big-O)?

```
def f(p,r):
  if p < r:
    q = (p + r) // 2
    f(p,q)
    f(q,r)
    for i in [p..r-1]:
      {some statement}
f(0,n)
```

$\longrightarrow$ $O(n \lg n)$

Searching concept

- n 개의 숫자가 있을 때 x 라는 값이 있는지 확인하는 알고리즘의 복잡도는? $O(n)$
- 65542 가 있는가?

```
x = [ 2347 , 13612 , 19386 , 26723 , 27999 ,
      40694 ,42533 , 65541 , 73593 , 96272 ]
```

$\longrightarrow$ Sorting 된 자료라면 원하는 값을 더 효과적으로 찾을 수 있다

$\longrightarrow$ $O(\lg n)$

Application of search

- Find the complexity of the following system (library)

```
for i in [0..n-1]:
  isbn = input_isbn()
  info = find(isbn)
  print(info)
```

- `input_isbn()` $\longrightarrow O(n)$

  $\longrightarrow \quad O(n^2)$

- If sorted, `input_isbn()` $\longrightarrow O(n)$

  $\longrightarrow \quad O(n \lg n)$

Improved search

- Find the complexity of the following system (library)

```
insertion_sort()
for i in [0..n-1]:
  isbn = input_isbn()
  info = find(isbn)
  print(info)
```

- Complexity: $O(n^2) + O(n \lg n) \longrightarrow O(n^2)$

$\longrightarrow$ We need better sorting algorithm

Mergesort

- 다음을 sorting 하시오
  - 1 3 5 7 9 2 4 6 8 10
  - 1 2 3 4 5 6 7 8 9 10
- 다음을 sorting 하시오
  - 1 3 5 9 11 2 4 6 8 10
  - 1 2 3 4 5 6 8 9 10 11

$\longrightarrow$ 이미 sorting 되어 있는 두 개의 merge

Mergesort

- Idea: If there is no element it is already sorted

```
def mergesort(A,p,r):
  if p < r:
    q = (p + q) // 2
    mergesort(A,p,q)
    mergesort(A,q+1,r)
    merge(A,p,q,r)
```

다음 중 가장 큰 숫자는?

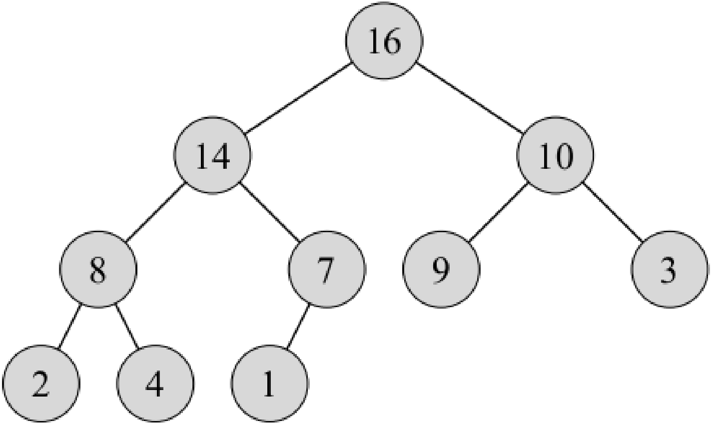

그림: Heap

새로운 개념: Heap

- Heap
  - 가장 꼭대기에 가장 큰 값이 올라와 있는 것 (The largest value resides on top)
  - 아래쪽에도 같은 원리가 반복 적용 되는 것 (The same rule applies to lower levels recursively)

Heap: 배열에 저장

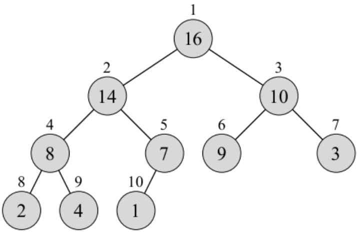

그림: Conceptual heap

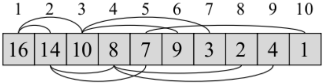

그림: Array implementation

Heap: Basic operation

$\text{PARENT}(i)$
1 **return** $\lfloor i/2 \rfloor$

$\text{LEFT}(i)$
1 **return** $2i$

$\text{RIGHT}(i)$
1 **return** $2i + 1$

그림: Basic operations

Idea: Partition

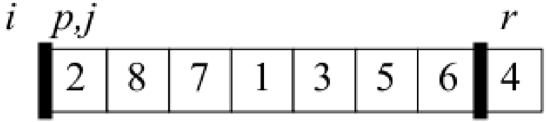

그림: Pivot

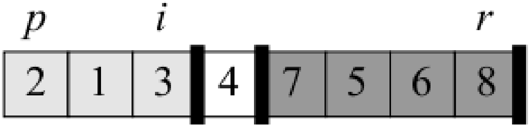

그림: Partition

Binary search

- 정렬되어 있다고 가정
- 찾고자 하는 값이 범위 안에 있는지 확인
- 없으면 중앙값과 비교
- 중앙값과 비교하여 왼쪽이나 오른쪽에서 다시 검색

Merge algorithm

- 두 개의 sort 된 list
  - 1 3 5 9 11
  - 2 4 6 8 10
- 알고리즘
  - 두 리스트의 가장 앞에 있는 값을 비교한다
  - 작은 쪽을 선택하여 새로운 list 에 넣는다
  - 이 작업을 한 쪽 리스트가 빌 때까지 반복한다
  - 남은 값들은 뒤에 넣는다

$\longrightarrow$ Linked list 를 이용한 sorting 도 가능

Merge algorithm

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

그림: Merge

Mergesort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

그림: Mergesort

Recursion and sorting

- Recursion is solving smaller problems with the same algorithm

```
def binary_search(a, x, lo, hi):
    if lo < hi:
        mid = (lo+hi)//2
        midval = a[mid]
        if midval < x:
            return binary_search(a, x, mid+1, hi)
        elif midval > x:
            return binary_search(a, x, lo, mid)
        else:
            return mid
    return -1
```

Recursion and sorting

- There are efficient sorting algorithms of complexity $O(n \lg n)$
- These sorting algorithms are best described using recursion

Common sorting algorithms

- Based on iteration $\longrightarrow$ Easy to understand
- Worst case complexity: $O(n^2)$

$\longrightarrow$ Paradigm change is needed

Heapify

- 두 개의 heap L, R, 새로운 값 $n \longrightarrow$ 새로운 heap

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

그림: Heapify

Heapsort

- 가장 큰 element 를 찾는다
- 제일 뒤로 보낸다
- 나머지에 대해서 반복한다

Heapsort

- Guaranteed $O(n \lg n)$
- Stable sorting

Quicksort algorithm

- 주어진 영역에서 pivot element 를 하나 선택 (임의로 선택)
- pivot element 보다 작은 것은 왼쪽에, 그렇지 않은 것은 오른쪽에
- 중앙에 pivot element 를 위치
- 이 과정을 재귀적으로 적용

Quicksort

- Most popular sorting algorithm but $O(n^2)$
- Worst cast could be easily avoided

$\longrightarrow$ Randomize

프로그래밍 연습

- Merge, Heapify, Partition 을 구현하시오.
  - 예시
    - 다음 입력에 대하여 Merge,Heapify,Partition 의 결과를 작성하시오.

```
10 20 30 15 25 35
```

Binary search

```
def binary_search(a, x, lo, hi):
    if lo < hi:
        mid = (lo+hi)//2
        midval = a[mid]
        if midval < x:
            return binary_search(a, x, mid+1, hi)
        elif midval > x:
            return binary_search(a, x, lo, mid)
        else:
            return mid
    return -1
```

$\longrightarrow$ Worst case complexity: $O(\log n)$

Binary search: Limitations

- Suppose searching for k values among Given n items,
- and k is quite small compared to n
  - Sorting $O(n^2)$, searching $O(\log n) \longrightarrow O(n^2)$
  - Sequential searching: $O(kn)$

$\longrightarrow$ We cannot decide which one is better

$\longrightarrow$ Look for better sorting algorithms

Mergesort

- Time complexity: $O(n \log n)$
- Applicable to Linked list

$\longrightarrow$ $O(n \log n)$ sorting algorithm for linked list

- Limitation: Need temporary memory

Rooted Binary Tree

- Root가 있고
- 루트와 연결된 link 가 left, right 에 하나씩 있고
- left 와 right 에 연결된 것은 rooted binary tree 이다

Algorithm: Partition


```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

그림: Partition

Quicksort


```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

그림: Quicksort

Binary search in C

- ex12.c

```
int binary_search(int ary[], int x, int lo, int hi)
{
  int mid, midval;
  if (lo < hi) {
    mid = (lo + hi)/2;
    midval = ary[mid];
    if (midval < x)
      return binary_search(ary, x, mid+1, hi);
    else if (midval > x)
      return binary_search(ary, x, lo, mid);
    else
      return mid;
  }
  return -1;
```

Binary search in Python

- ex11.py

```
def binary_search(a, x, lo, hi):
    if lo < hi:
        mid = (lo + hi)//2
        midval = a[mid]
        if midval < x:
            return binary_search(a, x, mid+1, hi)
        elif midval > x:
            return binary_search(a, x, lo, mid)
        else:
            return mid
    return -1
```

Binary search using iteration

```
def binary_search(a, x, lo=0, hi=None):
    if hi is None:
        hi = len(a)
    while lo < hi:
        mid = (lo+hi)//2
        midval = a[mid]
        if midval < x:
            lo = mid+1
        elif midval > x:
            hi = mid
        else:
            return mid
    return -1
```

Merge


```
def merge(tmp,A,p,q,r):
    for i in range(p,r):
        tmp[i] = A[i]
    i = p
    j = q
    while i < q and j < r:
        if tmp[i] < tmp[j]:
            A[p] = tmp[i]
            i = i + 1
        else:
            A[p] = tmp[j]
            j = j + 1
        p = p + 1
    while i < q:
        A[p] = tmp[i]
        i = i + 1
        p = p + 1
    while j < r:
        A[p] = tmp[j]
        j = j + 1
        p = p + 1
```

Mergesort

```
def mergesort(tmp, A,p,r):
    if p < r - 1:
        q = (p + r) // 2
        mergesort(tmp, A,p,q)
        mergesort(tmp, A,q,r)
        merge(tmp, A,p,q,r)
```

Python implementation


```
def parent(n):
    return (n-1)//2
def left(n):
    return 2*n+1
def right(n):
    return 2*n+2
```

Heapify


```
def heapify(A,i,heapsize):
    l = left(i)
    r = right(i)
    if l < heapsize and A[l] > A[i]:
        largest = l
    else:
        largest = i
    if r < heapsize and A[r] > A[largest]:
        largest = r
    if largest != i:
        A[i],A[largest] = A[largest],A[i]
        heapify(A,largest,heapsize)
```

Heapsort

```
def buildheap(A):
    for i in range(len(A)//2,0,-1):
        heapify(A,i-1,len(A))
def heapsort(A):
    buildheap(A)
    for i in range(len(A),1,-1):
        A[i-1],A[0] = A[0],A[i-1]
        heapify(A,0,i - 1)
```

Partition


```
def partition(A, p, r):
    x = A[r-1]
    i = p - 1
    for j in range(p,r-1):
        if A[j] <= x:
            i = i + 1
            A[i],A[j] = A[j],A[i]
    A[i+1],A[r-1] = A[r-1],A[i+1]
    return i+1
```

Quicksort


```
def quicksort(A,p,r):
    if p < r:
        q = partition(A,p,r)
        quicksort(A,p,q)
        quicksort(A,q+1,r)
```

Exercise

- Fill in the blank

```
def merge(tmp,A,p,q,r):
    for i in range(p,r):
        tmp[i] = A[i]
    i = p
    j = q
    while i < q and j < r:
        if _______________:
            A[p] = tmp[i]
            i = i + 1
        else:
            A[p] = tmp[j]
            j = j + 1
        p = p + 1
    while i < q:
        A[p] = ______
        _________
        p = p + 1
    while j < r:
        A[p] = ______
        _________
        p = p + 1
```

Other exercise

- Write an algorithm to sort in $O(n \lg n)$ time
- Show an example where Quicksort algorithm takes $O(n^2)$ time complexity

Wrap-up

- Given a sorted array, we could find an element in $O(\lg n)$ time even for the worst cases
- Sorting constitutes a basis of building efficient softwares
- Algorithms based on simple observation mostly results in $O(n^2)$
- But studying an abstract data structure called heap, we've come up with an $O(n \log n)$ algorithm, which is called heapsort
- Though quicksort looks efficient, the worst case time complexity of it is $O(n^2)$