Part II. Sorting and order statistics

Chapter 6. Heapsort, the use of priority queue

Chapter 7. Quicksort, analysis

Chapter 8. Sorting in linear time, lower bounds

Chapter 9. Medians and order statistics

Chapter 6. Heapsort

<u>heaps</u>

LEFT, RIGHT, PARENT

<u>heapsort</u>

BUILD-MAX-HEAP(A)

MAX-HEAPIFY(A, i)

HEAPSORT(A)

<u>heaps as priority queues</u>

HEAP-MAXIMUM(A)

HEAP-EXTRACT-MAX(A)

HEAP-INCREASE-KEY(a, I, key)

MAX-HEAP-INSERT(A, key)

PARENT(i) return $\lfloor i/2 \rfloor$

LEFT(i) return $2i$

RIGHT(i) return $2i + 1$

Max-heap property: every child node is less than or equal to it parent node.

```
HEAPSORT(A)
1. BUILD-MAX-HEAP(A)
2. for i <- length[A] down to 2
3.      do exchange A[1] <--> A[i]
4.         heap-size[A] <- heap-size[A] -1
5.         MAX-HEAPIFY(A, 1)
```

analysis of time

```
BUILD-MAX-HEAP(A)
1. heap-size[A] <-- length[A]
2. for i <-- length[A]/2 downto 1
3.   do MAX-HEAPIFY(A, i)

MAX-HEAPIFY(A, i)
1.l <-- LEFT[i]
2.r <-- RIGHT[i]
3.if l <=  heap-size[A] and A[l] > A[i]
4.    then largest <-- l
5.    else largest <-- i
6.if r <= heap-size[A] and A[r] > A[largest]
7.    then largest <-- r
8.if largest =\= i
9.   then exchange A[i] <-> A[largest]
10.       MAX-HEAPIFY(A, largest)
```

```
HEAP-MAXIMUM(A)
return A[1]
```

```
HEAP-EXTRACT-MAX(A)
if heap-size[A] <1
   then error "heap underflow"
max <-- A[1]
A[1] <-- A[heap-size[A]]
heap-size[A] <--  heap-size[A] -1
MAX-HEAPIFY(A, 1)
return max
```

Chapter 7. Quicksort

the basic idea of ”quicksort”

the algorithm details

average case running time (probabilistic analysis)

randomized algorithm

variants

divide-and-conquer for quicksort

divide: partition list $A[p, r]$ into two sublists $A[p, q-1]$ and $A[q+1, r]$ such that

(a) $A[i] \leq A[q]$ for all $i = p, \cdots, q-1$
(b) $A[i] > A[q]$ for all $i = q+1, \cdots, r$

conquer: sort $A[p, q-1]$ and $A[q+1, r]$ recursively.

combine: no further work is needed.

```
QUICKSORT(A, p, r)
1. if p< r
2.   then q <-- PARTITION(A, p, r)
3.        QUICKSORT(A, p, q-1)
4.        QUICKSORT(A, q+1, r)
```

1. If `p<=k<=i`, then `A[k]<=x`
2. If `i+1<=k<=j-1`, then `A[k]>x`
3. If `k=r`, then `A[k]=x`

```
PARTITION(A, p, r)
1. x <-- A[r]
2. i <-- p-1
3. for j <-- p to r-1
4.   do if A[j] <= x
5.        then i <-- i +1
6.             exchange A[i] <--> A[j]
7. exchange A[i+1] <--> A[r]
8. return i+1
```

example: Figure 7.1 (page 147)

Is there an easier way (not necessarily the most efficient way) to do partition?

analysis of performance

worst case partitioning: $T(n) = T(n-1) + \Theta(n)$

best case : $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil - 1) + \Theta(n)$

balanced partitioning:

$T(n) = T(9n/10) + T(n/10) + \Theta(n)$

using recursive tree method, it can be shown that

$T(n) = O(n \log_{10} n)$ and
$T(n) = \Omega(n \log_{10/9} n)$.

Therefore, $T(n) = \theta(nlogn)$.

average case performance

probabilistic analysis: assume that lists to be sorted are random lists.

An intuition that PARTITION should give a balanced partition *with a high probability* on a *random list*:

(1) Consider a list of n elements in which every element has equal chance to be in position i, $i = 1, 2, \ldots, n$. That is, for any position j and element $A[i]$,

$$Pr(rank(A[i]) = j) = 1/n$$

(2) Then the probability for any chosen pivot to partition the list into two lists of sizes $n/10$ and $9n/10$ is 80%

(3) We can say that most (i.e., 80%) of the time QUICKSORT runs $O(n \log n)$.

Analysis Method 1: taking the average

The pivot could be at any of these positions $1, 2, \ldots, n-1, n$. Then

$$T(n) = 1/n * [\Sigma_{j=2}^{n-1}(T(j-1) + T(n-j)) + 2T(n-1)] + cn$$

Consider two categories of partitions:

"good partition" $(n/4 < j \leq 3n/4)$

"bad partition" $(1 \leq j \leq n/4 \quad \text{or} \quad 3n/4 < j \leq n)$

$T(n)$ is then the sum of two terms (exercise).

One can use the substitution or recursive tree method to prove $T(n) = O(n \log n)$.

Analysis Method 2: randomized quicksort

```
RANDOMIZED-PARTITION(A, p, r)
1.  i <-- random(p, r)
2.  exchange A[r] <--> A[i]
3.  return PARTITION(A, p, r)
```

rename elements in A as

$z_1, \cdots, z_n$ with z_i being the ith smallest element, and
$Z_{ij} = \{z_i, \cdots, z_j\}$.

Define: $X_{ij} = 1$ iff z_i is compared to z_j

and $X = \Sigma_{i=1}^{n-1} \Sigma_{j=i+1}^{n} X_{ij}$ the number of comparisons.

$$E[X] = \Sigma_{i=1}^{n-1} \Sigma_{j=i+1}^{n} E[X_{ij}]$$
$$= \Sigma_{i=1}^{n-1} \Sigma_{j=i+1}^{n} Pr(z_i \text{ is compared to } z_j)$$

Event $\underline{z_i \text{ is compared to } z_j}$ occurs when z_i is chosen as a pivot or z_j is chosen to be a pivot. The chance of each element to be chosen is equally likely.

Note: that z_i or z_j is chosen has nothing do with any element outside of Z_{ij} being chosen. Therefore,

$Pr(z_i \text{ is compared to } z_j)$
$= 2/|Z_{ij}| = 2/(j - i + 1)$

Then $E[X] = \Sigma_{i=1}^{n-1} \Sigma_{k=1}^{n-i} 2/(k+1) = O(n \log_2 n)$

Chapter 8 Lower bounds and sorting in linear time

Deriving lower bounds

Comparison-based computational model

(1) Prove that "finding the max" among n elements needs at least $(n-1)$ comparisons.

(2) Prove that "sorting n elements" needs $\Omega(n \log n)$ comparisons.

sorting in linear time

A lower bound for sorting:

decision tree – a full binary tree (every node has zero or two children) modeling algorithms/computations

each internal node denotes $(x_i \leq x_j)$, with two outcomes

each leaf denotes a possible output of the algorithm

Claim 1: total number of leaves is $n!$.

Claim 2: the maximum number of leaves for a binary tree of height h is 2^h.

Theorem: Sorting needs $\Omega(n \log n)$ comparisons on comparison-based computation models.

Prove: The longest path from the root to a leave is $\Omega(\log n!)$. i.e., the number of comparisons needed in the worst case is $\Omega(\log(n!))$.

$n! = n(n-1)(n-n/2)(n-n/2-1)\cdots 2\times 1 \geq (n/2)^{n/2}2^{n/2-1}$

$\geq (n)^{n/2}/2.$

or by Stirling's formula:

$$n! = \sqrt{2\pi n}(n/e)^n(1+O(1/n))$$

$$\Omega(\log(n!)) = \Omega(n \log n)$$

Count Sort:

```
COUNTING-SORT(A, B, k)
for i <--0 to k  {k is the largest element}
  do C[i] <-- 0
for j = 1 to length[A]
  do C[A[j]] <-- C[A[j]] + 1
 {C[i] contains the number of elements = i}
for i <-- 1 to k
  do C[i]=C[i] + C[i-1]
 {C[i] contains the number of elements <= i}
for j <-- length[A] downto 1
 {either upwards or downwards is ok}
  do B[C[A[j]]] <-- A[j]
     C[A[j]] <-- C[A[j]] -1
```

example: A: 2 5 3 0 2 3 0 3   C: 2 0 2 3 0 1 C: 2 2 4 7 7 8

Analysis? $T(n) = O(k + n)$

Radix Sort:

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

```
RADIX-SORT(A, d)
1. for i <-- 1 to d {note the direction!)
2.   sort A on digit i
```

Lemma 8.4: Given n b-bit numbers and any positive $r \leq b$. RADIX-SORT uses $\Theta((b/r)(n + 2^r))$ time.

Proof: each number is of $\lceil b/r \rceil$ digits of r bits (binary bits) each.

Bucket Sort: assuming uniform distribution of inputs

```
BUCKET_SORT(A)
1. n <-- length[A]
2. for i <-- 1 to n
3.   do insert A[i] into list B[nA[i]]
4. for i <-- 0 to n-1
5.   do sort list B[i] with insertion sort
6. concatenate the list B[0], B[1], ..., B[n-1]
```

```
e.g.:  A:  .78 .17 .39 .26 .72 .94 .21 .12 .23 .68
       B:    0 /
             1 -> .12 -> .17
             2 -> .21 _> .23 -> .26
             3 -> .39
             4 /
             5 /
             6 -> .68
             7 -> .72 -> .78
             8 /
             9 -> .94
```

analysis? average time

Chapter 9. Medians and order statistics

The selection problem

Input: A set A of n (distinct) numbers and i, $1 \le i \le n$;

Output: $x \in A$, the ith smallest element in A.

Selection in expected linear time (but worst case $\Theta(n^2)$)

Selection in worst case linear time

Selection in expected linear time

```
RANDOMIZED-SELECT(A, p, r, i)
if p=r
 then return A[p]
q <- RANDOMIZED-PARTITION(A, p, r)
k <- q - p + 1
if i = k
  then return A[q]
else if i < k
    then return RANDOMIZED-SELECT(A, p, q-1, i)
else return RANDOMIZED-SELECT(A, q+1, r, i-k)
```

analysis?

worst case running time $\Theta(n^2)$.

average case (expected time): $E[T(n)]$

Note: $Pr(x_k \text{ is the pivot }) = 1/n$

Define $X_k = 1$ iff $A[p..q]$ has exactly k elements.

Then $E[X_k] = 1/n$ and $X_k = 1$ for exactly one value of k.

$T(n) \leq \Sigma_{k=1}^{n} X_k(T(\max\{k-1, n-k\}) + O(n))$

$E[T(n)] = \Sigma_{k=1}^{n} E[X_k \cdot T(\max\{k-1, n-k\})] + O(n)$

$= \Sigma_{k=1}^{n} 1/n \cdot E[T(\max\{k-1, n-k\})] + O(n)$

$\max\{k-1, n-k\} = k-1$ if $k > n/2$
$\max\{k-1, n-k\} = n-k$ if $k \leq n/2$

$E[T(n)] \leq 2/n \Sigma_{k=n/2}^{n-1} E[T(k)] + O(n)$

Solving the recurrence:

We have $E[T(n)] = O(n)$.

Selection in worst case linear time

deterministically finds a good partition:

(1) divide n elements into n/5 groups of 5 elements

(2) find the median of each group

(3) recursively find the median x of medians

(4) use x as the pivot

why this is a good partition?

Note: the number of elements $\leq x$ is at least:

$$3(1/2\lceil n/5 \rceil - 2) \geq 3n/10 - 6$$

similarly, the number of elements $\geq x$ is at least:

$$3(1/2\lceil n/5 \rceil - 2) \geq 3n/10 - 6$$

$T(n) \leq T(\lceil n/5 \rceil) + T(\lceil 7n/10 + 6 \rceil) + O(n)$

when $n \geq 140$