Design and Analysis of Algorithm

M. Sakthi Balan

Department of Computer Science & Engineering
LNMIIT, Jaipur

Contents

Algorithm 1.1: INSERTION-SORT(A)

$$
\textbf{for } j \leftarrow 2 \textbf{ to } \mathit{length}[A] \\
\textbf{do} \begin{cases} \mathit{key} \leftarrow A[j] \\ i \leftarrow j - 1 \\ \textbf{while } i > 0 \textbf{ and } A[i] > \mathit{key} \\ \quad \textbf{do} \begin{cases} A[i+1] \leftarrow A[i] \\ i \leftarrow i - 1 \end{cases} \\ A[i+1] \leftarrow \mathit{key} \end{cases}
$$

Algorithm 2.1: SELECTION-SORT(A)

for $i \leftarrow 1$ **to** $length[A] - 1$
do
- $min \leftarrow i$
- **for** $j \leftarrow i + 1$ **to** $length[A]$
  - **do**
    - **if** $A[j] < A[min]$
    - **then** $\{ min \leftarrow j$
- **if** $min\ != \ i$
  - **then**
    - $temp \leftarrow A[min]$
    - $A[min] \leftarrow A[i]$
    - $A[i] \leftarrow temp$

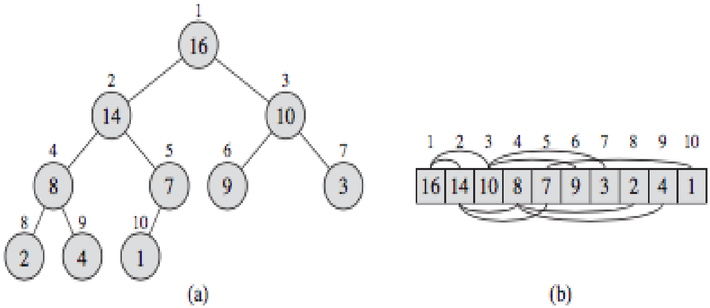
Algorithm 3.1: BUBBLE-SORT(A)

for $i \leftarrow 1$ **to** $length[A]$
do {
  for $j \leftarrow 1$ **to** $length[A] - i - 1$
  do {
    if $A[j] > A[j+1]$
    then {
      $temp \leftarrow A[j]$
      $A[j] \leftarrow A[j+1]$
      $A[j+1] \leftarrow temp$

Heapsort

- Heap data structure is an array object that can be viewed as a complete binary tree
- Tree is completely filled except possibly the lowest level
- Filled from left to right
- $length[A]$ denotes number of elements in A
- $heap-size[A]$ denotes the number of elements in the heap
- The root of the tree is $A[1]$
- For any i, $Parent[i]$, $Left[i]$ and $Right[i]$ can be easily computed

Heapsort



From CLRS

Heapsort

PARENT(i)
  return $\lfloor i/2 \rfloor$
LEFT(i)
  return $2i$
RIGHT(i)
  return $2i + 1$

Heap Property

$A[\text{PARENT}(i)] \geq A[i]$

LNMIIT

Heapsort

PARENT(i)

  return $\lfloor i/2 \rfloor$

LEFT(i)

  return $2i$

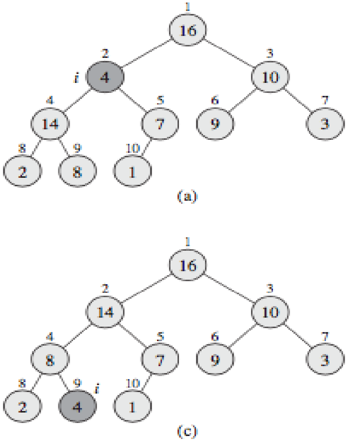
RIGHT(i)

  return $2i + 1$

Heap Property

$A[\text{PARENT}(i)] \geq A[i]$

Maintaining the Heap Property



From CLRS

Maintaining the Heap Property

Algorithm 4.1: HEAPIFY(A, i)

l← *Left*(i)
r← *Right*(i)
if l <= heap-size[A] **and** A[l] > A[i]
  then *largest* ← *l*
  else *largest* ← *i*
if r <= heap-size[A] **and** A[r] > A[largest]
  then *largest* ← *r*
if largest <> i
  then { *exchange* A[i], A[largest]
  HEAPIFY(A, *largest*)

Maintaining the Heap Property

When Heapify is called for *i* it is assumed that LEFT(I) and RIGHT(I) are already heaps. Time complexity of HEAPIFY(A,i) is $\mathcal{O}(\log n)$

Building a Heap

Algorithm 4.2: BUILD-HEAP(A)

heap-size[A] $\leftarrow$ length[A]
for $i \leftarrow \lfloor \mathit{length}[A]/2 \rfloor$ **downto** 1
  do HEAPIFY(A, i)

Building a Heap

Time complexity of BUILD-HEAP:

$$\sum_{h=0}^{\log n} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor \mathcal{O}(h) = \mathcal{O}\left(n \sum_{h=0}^{\log n} \frac{h}{2^h} \right)$$

But we know that

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

Hence we have

$$\sum_{h=0}^{\log n} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor \mathcal{O}(h) = \mathcal{O}(n)$$

Heapsort

Algorithm 4.3: HEAPSORT(A)

BUILD-HEAP(A)
for i $\leftarrow$ length[A] **downto** 2

$$\textbf{do} \begin{cases} \textit{exchange } \mathrm{A[1], A[i]} \\ \text{heap-size[A]} \leftarrow \text{heap-size[A]} - 1 \\ \textsc{Heapify}(A, 1) \end{cases}$$

Heapsort

Example: Sort $A = \{5, 13, 2, 25, 7, 17, 20, 8, 4\}$ using Heapsort

Priority Queues

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a *key*
A max-priority queue supports the following operations:

- $\textsc{Insert}(S, x)$
- $\textsc{Maximum}(S)$
- $\textsc{Extract-Max}(S)$
- $\textsc{Increase-Key}(S, x, k)$

Priority Queue

Schedule jobs on a shared computer

Priority Queue

HEAP-MAXIMUM(A)

1 **return** $A[1]$

From CLRS

Priority Queue

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

From CLRS

Priority Queue

```
HEAP-INCREASE-KEY(A, i, key)
  if key < A[i]
      error "new key is smaller than current key"
  A[i] = key
  while i > 1 and A[PARENT(i)] < A[i]
      exchange A[i] with A[PARENT(i)]
      i = PARENT(i)
```

From CLRS

Priority Queue

MAX-HEAP-INSERT(A, key)

1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$

From CLRS

Priority Queue

Building a heap using insertion function MAX-HEAP-INSERT

Quicksort

1. developed in 1959 by Tony Hoare while in the Soviet Union (Moscow State University)
2. worked in a project for the National Physical Laboratory
3. sort the words of Russian sentences prior to looking them up in a Russian-English dictionary
4. insertion sort was too slow for him
5. returned to England, implemented it in ALGOL that supports recursion and published in 1961 in ACM

Quicksort

Divide: Rearrange the array elements with respect to a pivot element. And using the pivot element break the array into two.

Conquer: Sort the two subarrays by recursively calling quicksort

Combine: Combine the sorted subarrays

Quicksort

$\text{QUICKSORT}(A, p, r)$

$\quad$ **if** $p < r$

$\qquad q = \text{PARTITION}(A, p, r)$

$\qquad \text{QUICKSORT}(A, p, q-1)$

$\qquad \text{QUICKSORT}(A, q+1, r)$

From CLRS

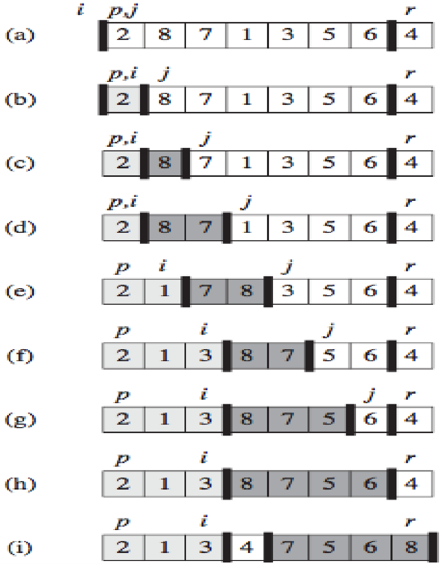
LNMIIT

Quicksort

```
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
```

From CLRS

Quicksort


(a) i p,j r
2 8 7 1 3 5 6 4
(b) p,i j r
2 8 7 1 3 5 6 4
(c) p,i j r
2 8 7 1 3 5 6 4
(d) p,i j r
2 8 7 1 3 5 6 4
(e) p i j r
2 1 7 8 3 5 6 4
(f) p i j r
2 1 3 8 7 5 6 4
(g) p i j r
2 1 3 8 7 5 6 4
(h) p i r
2 1 3 8 7 5 6 4
(i) p i r
2 1 3 4 7 5 6 8

Performance of Quicksort

Worst-case

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \Theta(n^2)
\end{aligned}
$$

Best-case

$$
\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n) \\
&= \Theta(n \log n)
\end{aligned}
$$

Average Case Analysis

Average-case is more closer to the best-case than the worst-case!

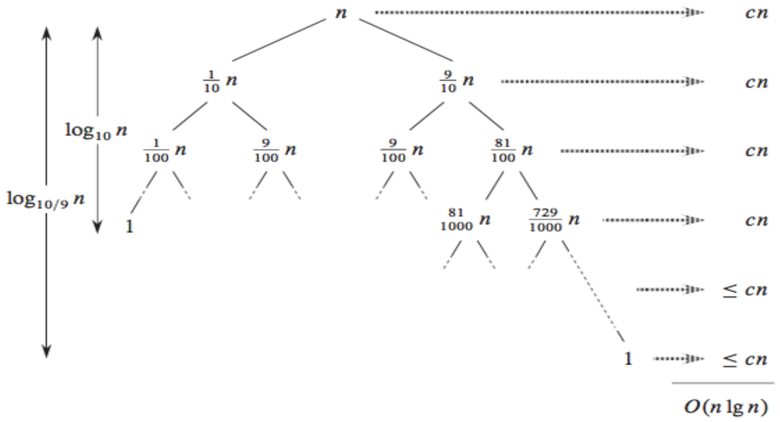
$$T(n) = T(9n/10) + T(n/10) + cn$$

Average Case Analysis

Average-case is more closer to the best-case than the worst-case!

$$T(n) = T(9n/10) + T(n/10) + cn$$

Quicksort



From CLRS

A randomized version of quicksort

Algorithm 6.1: $\textsc{Randomized-Partition}(A, p, r)$

i $\leftarrow$ $\textsc{Random}(p, r)$
exchange A[r] *and* A[i]
return $(\textsc{Partition}(A, p, r))$

Algorithm 6.2: $\textsc{Randomized-Quicksort}(A, p, r)$

if p < r
then

$$\textbf{do} \begin{cases} q \leftarrow \textsc{Randomized-Partition}(A, p, r) \\ \textsc{Randomized-Quicksort}(A, p, q-1) \\ \textsc{Randomized-Quicksort}(A, q+1, r) \end{cases}$$

Worst-case Analysis

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

We guess that $T(n) \leq cn^2$ and use the substitution method to prove it. Hence we obtain,

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n-q-1)^2) + \Theta(n) \\ &\leq c \cdot (n-1)^2 + \Theta(n) \\ &\leq c \cdot n^2 \end{aligned}$$

But since we already an example where $T(n) = \Omega(n^2)$ we get $T(n) = \Theta(n^2)$

Expected Running Time

Finding the expected number of comparisons done in PARTITION is the key!
Let $\{z_1, z_2, \ldots, z_n\}$ be the set where z_i is the ith smallest element
Let $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$
Now the question is when does z_i and z_j are compared?

Expected Running Time

X_{ij} is taken as 1 if z_i and z_j are compared and 0 if not
Then the total number of comparisons is given by

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Expected Running Time

Taking expectations on both the sides and applying linearity we get,

$$
\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} Pr[z_i \text{ compared with } z_j]
\end{aligned}
$$

Expected Running Time

$$
\begin{aligned}
Pr[z_i \text{ compared with } z_j] &= Pr[z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}] \\
&= Pr[z_i \text{ chosen }] + Pr[z_j \text{ chosen }] \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1}
\end{aligned}
$$

Expected Running Time

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
& \quad \text{Change of variable } k = j - i \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} \mathcal{O}(\log n) \\
&= \mathcal{O}(n \log n)
\end{aligned}
$$

Hoare's Partition

```
HOARE-PARTITION(A, p, r)
x = A[p]
i = p − 1
j = r + 1
while TRUE
    repeat
        j = j − 1
    until A[j] ≤ x
    repeat
        i = i + 1
    until A[i] ≥ x
    if i < j
        exchange A[i] with A[j]
    else return j
```

CLRS

From LNMIIT

Hoare's Partition

- Demonstrate HOARE-PARTITION on the array $A = (13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21)$
- Differences between HOARE-PARTITION and PARTITION
- What will be the performance of HOARE-PARTITION when the array elements are all same? Compare it with PARTITION
- Any volunteers to implement both the above algorithms and compare and contrast with respect to number of comparisons done in various inputs?

Mergesort

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
```

CLRS

From LNMIIT

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)
```

From CLRS

Other Sorting Methods

- Comparison sorts – sorts by comparing pair of elements
- Other sorting – counting sort, radix sort and bucket sort

Lower bound for sorting

Given any input sequence

$$(a_1, a_2, \ldots, a_n)$$

we check either $a < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i > a_j$, or $a_i \geq a_j$

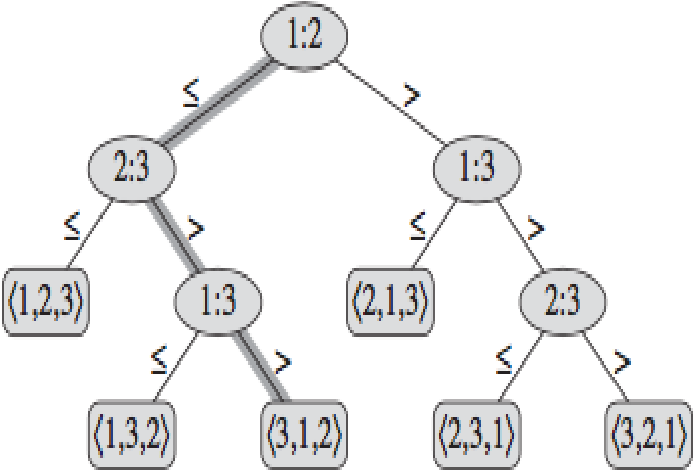
Lower bound for sorting

The Decision-Tree Model

Suppose there are n elements to be sorted. Build the decision tree like below:

- Each internal node is represented by $i : j$ for $1 \leq i, j \leq n$
- Each leaf is represented by a permutation of the sequence $(1, 2, \ldots, n)$

Decision Tree Model



Lower bound for sorting

Theorem

Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case

Lower bound for sorting

Possible sequences = $n!$
Number of leaves in a binary tree with height $h = 2^h$

$$\begin{aligned} n! &\leq 2^h \\ \log(n!) &\leq h \\ h &\geq \log(n!) \\ h &= \Omega(n \log n) \end{aligned}$$

Counting Sort

- Counting sort determines, for each input element x, the number of elements less than x
- Use the above information to place element x directly into its position in the output array

Counting Sort

Three arrays needed:

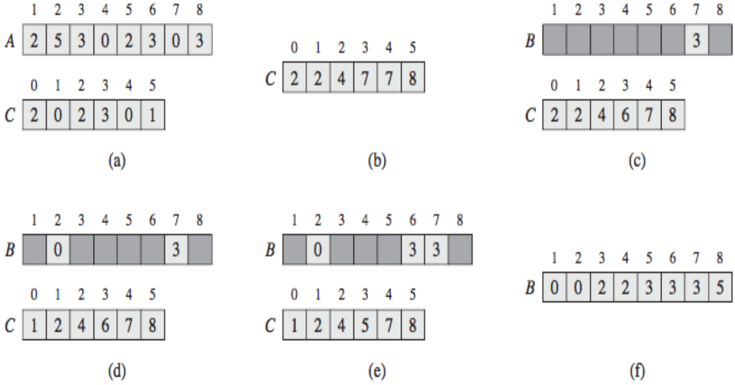
- $A[1:n]$ is the input array
- $B[1:n]$ that holds the sorted output array
- $C[0:k]$ that provides temporary storage

Counting Sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

From CLRS

Counting Sort



From CLRS

Counting Sort

Counting sort is Stable!

- Illustrate the operation of COUNTING-SORT on the array $A = (6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2)$
- Prove that COUNTING-SORT is stable
- Suppose that we rewrite the for loop header in line 10 of the COUNTING-SORT will the algorithm still work? Is it stable?

LNMIIT

Counting Sort

Counting sort is Stable!

- Illustrate the operation of COUNTING-SORT on the array $A = (6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2)$
- Prove that COUNTING-SORT is stable
- Suppose that we rewrite the for loop header in line 10 of the COUNTING-SORT will the algorithm still work? Is it stable?

Radix Sort

1. Radix sort is like sorting with respect to a column
2. RadixSort solves the sorting problem somehow "counter-intuitively", by starting with the least significant digit
3. Each element has d digits, where 1 is the lowest order digit, and d the highest-order digit

Radix Sort

Algorithm

RadixSort(A,d):
for i = 1 to d
Use a stable sort to sort array A on digit i

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

From CLRS

Radix Sort

Given n b-bit numbers and any positive integer $r \le b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n + 2^r))$ time if the stable sort it uses takes $\Theta(n + k)$ time for inputs in the range 0 to k.

Proof For a value $r \le b$, we view each key as having $d = \lceil b/r \rceil$ digits of r bits each. Each digit is an integer in the range 0 to $2^r - 1$, so that we can use counting sort with $k = 2^r - 1$. (For example, we can view a 32-bit word as having four 8-bit digits, so that $b = 32$, $r = 8$, $k = 2^r - 1 = 255$, and $d = b/r = 4$.) Each pass of counting sort takes time $\Theta(n + k) = \Theta(n + 2^r)$ and there are d passes, for a total running time of $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. ■

From CLRS

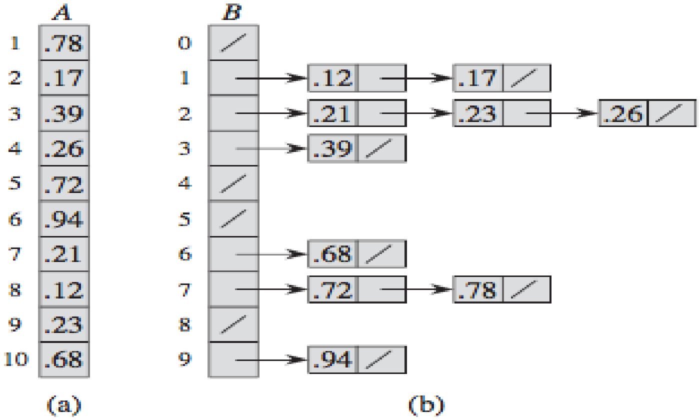
Bucket Sort

1. Assumes that the input is drawn from a uniform distribution
2. Assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval $[0, 1)$
3. Bucket sort divides the interval $[0, 1)$ into n equal-sized buckets, and then distributes the n input numbers into the buckets
4. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each

Bucket Sort

1. Input is n-element array A and that each element $0 \leq A[i] < 1$
2. An auxiliary array $B[0..n-1]$ of linked lists (buckets)

Bucket Sort



From CLRS

Bucket Sort

```
BUCKET-SORT(A)
let B[0..n − 1] be a new array
n = A.length
for i = 0 to n − 1
        make B[i] an empty list
for i = 1 to n
        insert A[i] into list B[⌊n A[i]⌋]
for i = 0 to n − 1
        sort list B[i] with insertion sort
concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

From CLRS