Introduction to Algorithms - Reading Notes & Selected Solutions

Nimrod Shneor

November 13, 2019

Solutions to selected exercises

Chapter 2

2.1-1

$$
\begin{aligned}
A &= [31, 41, 59, 26, 41] \\
A &= [31, 41, 59, 26, 41] \\
A &= [31, 41, 26, 59, 41] \\
A &= [31, 26, 41, 59, 41] \\
A &= [26, 31, 41, 59, 41] \\
A &= [26, 31, 41, 41, 59]
\end{aligned}
$$

2.1-4

Algorithm 1 BinaryAddition(A,B,n)

```
$carry \leftarrow 0$
**for** $i \leftarrow 1$ to $n$: **do**
    $C[i] \leftarrow (A[i] + B[i] + C[i])\, mod 2$
    $carry \leftarrow A[i] * B[i]$
**end for**
$C[i + 1] \leftarrow carry$
return $C$
```

- Input: two n-bit numbers A, B.
- Output: the sum of A, B - an $n + 1$-bit number.

2.2-3 Define X = The number of elements checked in a "brute force" linear search.

Than $X \in \{1...n\}$ and the average number of elements checked in a linear search is exactly:

$$E[X] = \sum_{i=1}^{n} \frac{1}{n} i = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n(n-1)}{2n} = \Theta(n)$$

The worst case is where the last element of the array is the one searched for - resulting in an $\Theta(n)$ run time.

2.3-3

$$T(n) = \begin{cases} 2 & n = 2 \\ 2T(\frac{n}{2}) + n & n > 2 \end{cases}$$

Q: Proof by induction that if n is an exact power of two (that is $n = 2^k$ for some constant $k \geq 1$) than $T(n) = nlogn$.

Proof: By induction.

Base case: for $k = 1$ than $T(n) = 2 = 2log2 = nlogn$
Assumption: Assume the above holds for all integers up to $k > 1$.
Induction step: We now prove the statement for $n = 2^{k+1}$.
Plugging in to the formula

$$\begin{aligned} T(n) = 2T(\frac{n}{2}) + n = 2T(\frac{2^{k+1}}{2}) + 2^{k+1} \\ = 2T(2^k) + 2^{k+1} = 2 * 2^k log2^k + 2^{k+1} \\ = k2^{k+1} + 2^{k+1} = (k+1)2^{k+1} = 2^{k+1}log2^{k+1} \\ = nlogn \end{aligned}$$

□

2.3-5

Algorithm 2 BinarySearch(A,x)

```
$l \leftarrow 0$
$r \leftarrow length(A)$
**while** $l < r - 1$ **do**
  **if** $x = A[\frac{l+r}{2}]$ **then**
    $\frac{l+r}{2}$
  **end if**
  **if** $x > A[\frac{l+r}{2}]$ **then**
    $l = A[\frac{l+r}{2}]$
  **else**
    $r = A[\frac{l+r}{2}]$
  **end if**
**end while**
return -1
```

At each iteration of the while loop the distance between the two pointers - l, r - is halfed until the element is found or we return -1. The while loop will terminate once the two pointers are at distance two at which point either the element x is found or the loop will terminate. Thus the distance between the two pointers at each iteration i is percisly $\frac{length(A)}{2^i} = \frac{n}{2^i}$

At the time of the termination the distance between the two pointers is two, thus -

$$\begin{aligned} \frac{n}{2^i} &= 2 \\ n &= 2^{i+1} \\ log(n) &= i + 1 \\ log(n) - 1 &= i \\ \Theta(log(n)) &= i \end{aligned}$$

2-1

Algorithm 3 FindSum(A,x)

2.3-7

```
$B \leftarrow MergeSort(A)$
$l \leftarrow 0$
$r \leftarrow length(B)$
**while** $l < r$ **do**
  **if** $B[l] + B[r] == x$ **then**
    return true
  **else if** $B[l] + B[r] < x$ **then**
    $l \leftarrow l + 1$
  **else**
    $r \leftarrow r - 1$
  **end if**
**end while**
return false
```

a Given $\frac{n}{k}$ lists each of size k. Applying *InsertionSort* to each list seperatly yealds worst-case runtime of $\Theta(k^2)$. Doing this for all $\frac{n}{k}$ lists yealds an $\Theta(\frac{n}{k}k^2) = \Theta(nk)$ runtime algorithm.

b

Algorithm 4 ModifiedMergeSort(A)

```
Split $A$ to form $S = [A_1, .., A_{\frac{n}{k}}]$ array of sub-arrays of size $k$
**for** $i \leftarrow 1$ to $\frac{n}{k}$ **do**
  $A_i \leftarrow InsertionSort(A_i)$
**end for**
**while** $|S| > 1$ **do**
  $l \leftarrow 1$
  $r \leftarrow length(S)$
  $S' \leftarrow \Phi$
  **while** $l < r$ **do**
    $S' \leftarrow S' \bigcup Merge(A_l, A_r)$
    $l \leftarrow l + 1$
    $r \leftarrow r - 1$
  **end while**
  $S \leftarrow S'$
**end while**
```

We prove that at each iteration of the outer while loop the size of $|S|$ is $\frac{n}{2^i k}$.
Proof: By induction,
Base $i = 1$: In the first iteration we set l and r to hold the two opposit ends of S, at each iteration we merge two subsets and continue so on until $l = r$ or $l > r$ (depending on the number of subsets) because at each iteration we merged two subsets the number of iterations of the inner loop is percisly $\frac{n}{2k}$.

Step: Assume that the number of subsets in $|S|$ is $\frac{n}{2^i k}$ at iteration i next we prove that at iteration $i+1$ the above statement holds.

Again, from the same argument for the base case - at each iteration of the inner loop the number of elements decrease by two the number of iterations of the inner loop is $\frac{n}{2^{i+1}k}$ yeilding that number of subsets.

The outer loop will terminate once $|S| = 1$, that is -

$$\begin{aligned} \frac{n}{2^i k} &= 1 \\ \frac{n}{k} &= 2^i \\ log(\frac{n}{k}) &= i \end{aligned}$$

At each iteration we perform $\frac{n}{2^i k}$ merges each runs in $\Theta(2^i k)$ for a total of $\Theta(n)$, Thus the total running time of the while loop is $\Theta(nlog(\frac{n}{k}))$

All together we get $\Theta(nlog(\frac{n}{k}) + nk)$.

c If one chooses $k = 1$ we get percisly $MergeSort$.

If one chooses $k = n$ we get percisly $InsertionSort$, Thus the choice of k needs to be as close as possible to one. If we choose $k = \Theta(1 - \frac{1}{n}) = \Theta(\frac{n-1}{n})$ which asymptotically is close to one we get -

$$\begin{aligned} T(n) &= \frac{n(n-1)}{n} + nlog(\frac{n}{\frac{n-1}{n}}) \\ &= n - 1 + nlog(\frac{n^2}{n-1}) \\ &\approx \Theta(nlogn) \end{aligned}$$

d In practice one can simply use $MergeSort$ or if one had to use the modified version, use smaller values of k checking these values "brute force".

2-4

a The inversions of $[2, 3, 8, 6, 1]$ are

$$(8, 6), (8, 1), (6, 1), (3, 1), (2, 1)$$

b The permutation τ of the set $\{1, .., n\}$ with the most inversions is $[n, n-1, n-2, ..., 1]$ it has $(n-1)+(n-2)+...+1 = \frac{n(n-1)}{2} = \Theta(n^2)$ inversions.

c We prove the following statement (x, y) is in the set of inversions of $S \iff$ its is switched in some iteration of the while loop in the $InsertionsSort$ algorithm.

$\Longleftarrow$ The pair $(x, y) = (A[i], A[j])$ is switched in some iteration of the *InsertionSort* algorithm, therefor by the loop definition $j = i + 1$ and $y < x$, therefor (x, y) is in the inversions set.

$\Longrightarrow$ $(x, y) = (A[i], A[j])$ are in the inversion set of A. we will prove the following Lemma:

Lemma: if $(x, y) = (A[i], A[j])$ are in the inversion set of A than for any integer $i < k \leq j$ the element $A[k]$ is also in the inversion set.

Proof: by induction on the distance between i and j.

Base: $j - i = 1$. Than by defition (x, y) are in the inversion set.

Step: Assume i and j are k elements apart and we prove the statement for i and j at distance $k + 1$. Assume by contradiction that $A[i + k]$ is not an inversion, therfor it is larger than any element that came before it - in particular $A[i]$, For otherwise it would be an inversion. If $A[i + k]$ is larger than $A[j]$ than the pair $(A[i + k], A[j])$ is an inversion for j and i are at distance $k + 1$. If $A[k + i]$ is smaller than $A[j]$ than the pair $(A[i + k], A[i])$ is an inversion since $A[i] > A[j] > A[i + k]$ by the assumption the $(A[i], A[j]) = (x, y)$ is in the inversion set. $\square$

By the Lemma any element between $A[i]$ and $A[j]$ are in the inversion set, that means that in the inner loop of *InsertionSort* all of those elements will be switched in the inner loop, including (x, y).

Conclusion: The number of elements in the inversion set of A is precisly the number of iterations of the inner loop of *InsertionSort*. In other words, the run time of *InsertionSort* is $\Theta(|S|)$.

Chapter 5

5.1-2

Algorithm 5 Random(a,b)

```
if $a = b$ then
  a
end if
while $a < b$ do
  if $Random(0, 1) > 0$ then
    $Random(a, \frac{a+b}{2})$
  else
    $Random(\frac{a+b}{2}, b)$
  end if
end while
```

The runtime of the above algorithm is $O(log(\frac{b-a}{2}))$.

6.1-1 The maximum number of elements in a heap of height h is $2^0+2^1+ ... + 2^h = \sum_{i=1}^{h} 2^i = \frac{2^{h+1}-1}{2-1} = 2^{h+1}-1$. The minimum number of elements occures when there is percisly one leaf node (i.e. the bottom level of the binary-tree is empty but one element), meaning:

$$2^0+2^1+...2^{h-1}+2 = \sum_{i=0}^{h-1} 2^i + 1 = \frac{2^h-1}{2-1}+1 = 2^h$$

6.1-2 Proof by Induction:

Base: $n=1$, A single node heap is at height $0 = log(1) = log(n)$

Step: We assume the statement is correct for $n=k-1$ and prove for a heap of size $n=k$. Consider the last element of the heap $A[k]$, by the definition of the heap, its parent is the element $A[\frac{k}{2}]$. By the induction step the heap $A[1...\frac{k}{2}]$ is of height $log(\frac{k}{2}) = log(k) - log(2) = log(k) - 1$. Thus, the height of the heap $A[1...k]$ has one more layer than $A[1...\frac{k}{2}]$, that is $log(k)$.

□

6.1-7 We prove the counter-positive statement, that is, if a node is indexed by $i \in \{1...\frac{n}{2}\}$ in the array representation of the heap than it is **not** a leaf node.

By contradiction, assume it was indexed by $i \in \{\frac{n}{2}+1, ..., n\}$ than, either one of his children had to be at some index k such that

$$k \geq 2i \geq 2(\frac{n}{2}+1) \geq n$$

Therefor it exceeds the size of the heap - contradiction.

Conclusion: a node is indexed by $i \in \{\frac{n}{2}+1, ...n\} \iff$ it is a leaf node in the heap.

Chapter 6

6-2

a One can represent a $d-ary$ heap using the following structure: for every non leaf node indexed by i in the array A, its childeren are indexed in the array at postiions $Children(i) = A[i*d+1, ..., i*d+d]$.

b The height of a $d-ary$ heap is $O(log_d n)$.

c First we need to modify $MAX-HEAPIFY$ for a $d-ary$ heap:

Once the new heapify is defined we can use the same procedure $Extract-Max$ as defined in the book except use our $d-ary\,MAX-HEAPIFY$ in line 6.

Algorithm 6 d-ary MAX-HEAPIFY(A,i)

```
$Children \leftarrow Children(i)$
$largest \leftarrow i$
for $j$ in 1...$d$: do
  if $A[i] > A[largest]$ then
    $largrest \leftarrow i$
  end if
end for
if $i \neq largest$ then
  $A[i] \leftrightarrow A[largest]$
  $d - ary\,MAX - HEAPIFY(A, largest)$
end if
```

Chapter 7

7.2-5 When the partition procedure produces a split at each level of proportion α to $1-\alpha$ for some costant $0 < \alpha \leq \frac{1}{2}$ we can write the runtime of *QuickSort* as the following recurrence:

$$T(n) = T(\alpha n) + T([1-\alpha]n) + \Theta(n)$$

At each level of the recurrence each input is again split in such manner, thus, the minimum depth of a leaf in the reccurrence tree occurres when the "smaller" split of the two is called recursivly until reaching the termination condition. That is, precisly when the size of the input is one -

$$\begin{aligned} n\alpha^h &= 1 \\ \alpha^h &= \frac{1}{n} \\ h &= log_\alpha(n) \\ h &= \frac{log(\frac{1}{n})}{log(\alpha)} \\ h &= -\frac{log(n)}{log(\alpha)} \end{aligned}$$

$\square$.

7.2-6 For a random input array of size n and a constant $0 \leq \alpha \leq \frac{1}{2}$ the number of elements which, if placed at the end of the array, produce a split more balanced that $1-\alpha$ to α is at most $n * 2 * (\frac{1}{2} - \alpha)$ - those are the elements which are closer to the median of the array (which produces a balanced split) than the $\alpha * 100$-precentile. Thus the probability for a random array of size n producing a split more balanced than α to $1-\alpha$ is at most

$$\frac{(n-1)! * n * 2 * (\frac{1}{2} - \alpha)}{n!} = \frac{n!}{n!} 2 * (\frac{1}{2} - \alpha) = 1 - 2\alpha$$

12-2.5 We proof by contradiction: Given a node v in the tree T such that v has two children $v.left$, $v.right$. We assume v's successor has a $left$ child. Since v has a right child than his successor is in the subtree rooted in $v.right$. Denote v's successor by z. On one hand, by the definition of the successor $v \leq z$, on the other hand by the definition of the $Binary\,Search\,Tree$ property $z.left \leq z$ and from the same argument, since $z.left$ is a part of the subtree rooted in $v.right$ it holds that $v \leq z.left$. Combining the two inequalities -

$$v \leq z.left \leq z$$

Thus, $z.left$ is v's successor - contradiction. $\square$.