1 Algorithm Analysis

1.1 Searching

Aside 1.1.1 (Unordered List) ***Linear Search:*** *W(n) = n.*

Aside 1.1.2 (Ordered List) ***Binary Search:*** *W(n) = 1 +* $\lfloor \log n \rfloor$

- *Binary tree of depth d has* $n \leq 2^{d+1} - 1$ *nodes*
- *Minimality of Binary Search = Depth of Tree =* $\lceil \log (n+1) \rceil$

1.2 Orders

Definition 1.2.1 *Let* $f, g : \mathbb{N} \to \mathbb{R}^+$.

- *f is O(g) iff* $\exists m \in N, \exists c \in \mathbb{R}^+$ *such that*

$$\forall n \geq m.[f(n) \leq c.g(n)]$$

- *f is* θ*(g) iff f is O(g) and g is O(f)*

Theorem 1.2.2 (Master Theorem)

$$T(n) = aT(n/b) + f(n)$$

Where $E = \frac{\log a}{\log b}$,

1. *If* $n^{E+\epsilon} = O(f(n))$ *for some* $\epsilon > 0$, $T(n) = \Theta(f(n))$ *— first level*
2. *If* $f(n) = \Theta(n^E)$, $T(n) = \Theta(f(n) \log n)$ *— every level*
3. *If* $f(n) = O(n^{E-\epsilon})$ *for some* $\epsilon > 0$, $T(n) = \Theta(n^E)$ *— base level*

1.3 Sorting

Definition 1.3.1 (Balanced Trees)

- *Total path length of a tree is the sum of the depths of all leaf nodes*
- *A tree of depth d is balance if every leaf is at depth d or d-1*
- *If a tree is unbalanced then we can find a balanced tree with the same number of leaves without increasing the total path length.*

Aside 1.3.2 (Decision Tree)

- *Binary tree of depth d has* $l \leq 2^d$ *leaves*
- *Sorting decision tree must have n! leaves*
- *Sorting by comparison must perform* $\lceil \log (n!) \rceil$ *comparisons*

- *Average of $\lfloor \log(n!) \rfloor$ comparison (balanced trees of depth d or d-1)*

Algorithm 1.3.3 (Insertion Sort) *Insert L[i] into L[0..i-1] in correct position. Then L[0..i] is sorted. $W(n) = n^2$.*

Algorithm 1.3.4 (Merge Sort) *Divide roughly into 2. Sort each half. Merge the two halves. $W(n) = n - 1 + W(\lceil \frac{n}{2} \rceil) + W(\lfloor \frac{n}{2} \rfloor) = n \log n - n + 1 = n \log n$.*

Algorithm 1.3.5 (Quicksort) *Split around first element then sort two sides recursively.*
$W(n) = \frac{n(n-1)}{2}$. $A(n) = n - 1 + \frac{1}{n} \sum_{s=1}^{n} (A(s-1) + A(n-s)) = n \log n$.

- *$left < i \leq j + 1$*
- *$j \leq right$*
- *if $left \leq k < i$ then $L[k] \leq d$*
- *if $j < k \leq right$ then $L[k] > d$*

```
Algorithm Split(left,right): # left < right
d = L[left] # pivot
i = left + 1 ; j = right
while i <= j:
    if L[i] <= d:
        i=i+1
    else:
        Swap(i, j) ; j = j - 1
Swap(left, j) ; return j
```

Algorithm 1.3.6 (Heapsort) *O(nlogn): n elements, logn deleteMax*

```
Build Max Heap H out of array E

SchemeHeapSort(H):
    for i = n to 1:
        E[i] = deleteMax(H)
    deleteMax(H):
        copy element at last node into root node
        remove last node
        fixMaxHeap(H)

ArrayHeapSort(E,n):
    heapsize = n
    while heapsize > 1:
        swap(1, heapsize)
        heapsize--
        fixMaxHeap(1, heapsize)
```

Algorithm 1.3.7 (Heaps) *Build Heap* $= \Theta(n)$

```
buildMaxHeap(H):
    if H not a leaf:
        buildMaxHeap(left subtree of H)
        buildMaxHeap(right subtree of H)
        fixMaxHeap(H)

fixMaxHeap(H):
    if H is not leaf:
        largerSubHeap = left/right subheap with larger root
        if root(H).key < root(largerSubHeap).key:
            swap elements at root(H) and root(largerSubHeap)
            fixMaxHeap(largerSubHeap)

insert(Q,x): # heap as array indexed at 1
    heapsize = heapsize + 1
    E[heapsize] = x
    percolateUp(heapsize)

percolateUp(c):
    if c>1:
        parent = floor(c/2)
        if E[c].key > E[parent.key]:
            swap(c, parent)
            percolateUp(parent)
```

1.4 Dynamic Programming

1.4.1 Top Down

```
wb1(s):
    if len(s) == 0:
        return true
    else:
        for i = 0 to len(s) - 1:
            if indict(s[i:]):
                if wb1(s[:i]):
                    return true
    return false
```

1.4.2 Memoised Top Down

```
memo = {}
wb2(s):
    if len(s) == 0:
        return true
    else:
        for i = 0 to len(s) - 1:
            if indict(s[i:]):
                if memo[s[:i]] undefined:
                    memo[s[:i]] = wb2(s[:i])
                if memo[s[:i]]:
                    return true
    return false

memo = {}
ws = ws2(s)
if ws >= 0:
    return ws
else:
    return ‘no possible splitting’

procedure ws2(s):
    if len(s) == 0:
        return 0
    else:
        bestscore = -1
        for i = 0 to len(s)-1:
            wordscore = score(s[i:])
            if wordscore > 0:
                if memo[s[:i]] undefined:
                    memo[s[:i]] = ws2(s[:i])
                if memo[s[:i]] >= 0 and
                memo[s[:i]] + wordscore > bestscore:
                    bestscore = memo[s[:i]]+ wordscore
        return bestscore
```

1.4.3 Bottom Up

```
wb3(s):
    n = len(s)
    wb[0] = true
    for i = 1 to n:
        wb[i] = false
        for j = 0 to i - 1:
            if wb[j] and indict(s[j : i]):
                wb[i] = true
                break
    return wb[n]

ws3(s):
    n = len(s)
    ws[0] = 0
    for i = 1 to n:
        ws[i] = -1
        for j = 0 to i - 1:
            wordscore = score(s[j : i])
            if ws[j] >= 0 and wordscore > 0:
                if ws[j] + wordscore > ws[i]:
                    ws[i] = ws[j] + wordscore
if ws[n] >= 0:
    return ws[n]
else:
    return ‘no possible splitting’
```