Algorithms Midterm Cheat Sheet

Recurrences

Master Theorem

- If $af(n/b) = \kappa f(n)$ for some constant $\kappa < 1$, then $T(n) = \Theta(f(n))$.
- If $af(n/b) = Kf(n)$ for some constant $K > 1$, then $T(n) = \Theta(n^{\log_b a})$.
- If $af(n/b) = f(n)$, then $T(n) = \Theta(f(n) \log_b n)$.

Annihilators

Operator	Functions annihilated
$\mathbf{E} - 1$	α
$\mathbf{E} - a$	αa^n
$(\mathbf{E} - a)(\mathbf{E} - b)$	$\alpha a^n + \beta b^n$
$(\mathbf{E} - a_0)(\mathbf{E} - a_1) \cdots (\mathbf{E} - a_k)$	$\sum_{i=0}^{k} \alpha_i a_i^n$
$(\mathbf{E} - 1)^2$	$\alpha n + \beta$
$(\mathbf{E} - a)^2$	$(\alpha n + \beta)a^n$
$(\mathbf{E} - a)^2(\mathbf{E} - b)$	$(\alpha n + \beta)a^b + \gamma b^n$
$(\mathbf{E} - a)^d$	$(\sum_{i=0}^{d-1} \alpha_i n^i)a^n$

If X annihilates f, then X also annihilates $\mathbf{E}f$.
If X annihilates both f and g, then X also annihilates $f \pm g$.
If X annihilates f, then X also annihilates αf, for any constant α.

If X annihilates f and Y annihilates g, then XY annihilates $f \pm g$.

Tower of Hanoi

Vanilla

```
function HANOI(n, src, dst, tmp)
    if n > 0 then
        HANOI(n − 1, src, tmp, dst)
        move disk n from src to dst
        HANOI(n − 1, tmp, dst, src)
    end if
end function
```

Variant 1

Every move must involve peg 0; that is, we are forbidden from directly moving a disk from the source peg to the destination peg.

```
function HANOIVARIANT1(n, src, dst, tmp)
    if n > 0 then
        HANOIVARIANT1(n − 1, src, dst, tmp)
        move disk n from src to tmp
        HANOIVARIANT1(n − 1, dst, src, tmp)
        move disk n from tmp to dst
        HANOIVARIANT1(n − 1, src, dst, tmp)
    end if
end function
```

Variant 2

We are only allowed to move counterclockwise; we can move from the source peg to the temp peg to the destination peg back to the source peg, but not the reverse order.

```
function HANOIVARIANT2(n, src, dst, tmp)
    if n = 1 then
        move disk n from src to tmp
        move disk n from tmp to dst
    else if n > 0 then
        HANOIVARIANT2(n − 1, src, dst, tmp)
        move disk n from src to tmp
        HANOIVARIANT2(n − 2, dst, tmp, src)
        move disk n − 1 from dst to src
        HANOIVARIANT2(n − 2, tmp, src, dst)
        move disk n from tmp to dst
        HANOIVARIANT2(n − 1, src, dst, tmp)
    end if
end function
```

Sorting

Merge Sort

```
function MERGESORT(A[1..n])
    if n > 1 then
        m ← ⌊n/2⌋
        MERGESORT(A[1..m])
        MERGESORT(A[m + 1..n])
        MERGE(A[1..n], m)
    end if
end function
function MERGE(A[1..n], m)
    i ← 1
    j ← m + 1
    for k ← 1, n do
        if j > n then
            B[k] ← A[i]
            i ← i + 1
        else if i > m then
            B[k] ← A[j]
            j ← j + 1
        else if A[i] < A[j] then
            B[k] ← A[i]
            i ← i + 1
        else
            B[k] ← A[j]
            j ← j + 1
        end if
    end for
    for k ← 1, n do
        A[k] ← B[k]
    end for
end function
```

Quicksort

```
function QUICKSORT(A[1..n])
    if n > 1 then
        Choose a pivot element A[p]
        r ← PARTITION(A, p)
        QUICKSORT(A[1..r − 1])
        QUICKSORT(A[r + 1..n])
    end if
end function
function PARTITION(A[1..n], p)
    swap A[p] ↔ A[n]
    i ← 0
    j ← n
    while i < j do
        repeat i ← i + 1 until i ≥ j or A[i] ≥ A[n]
        repeat j ← j − 1 until i ≥ j or A[j] ≤ A[n]
        if i < j then
            swap A[i] ↔ A[j]
        end if
    end while
    swap A[i] ↔ A[n]
    return i
end function
```

Subsequences

Longest Common Subsequence

```
function LCS(A[1..m], B[1..n])
    if m = 0 or n = 0 then
        return 0
    else
        a ← LCS(A[2..m], B[1..n])
```

```
        b ← LCS(A[1..m], B[2..n])
        c ← 0
        if A[1] = B[1] then
            c ← 1+ LCS(A[2..m], B[2..n])
        end if
        r ← the maximum of a, b, and c
    end if
end function
```

Longest Oscillating Subsequence

```
function LOS(X[1..n])
    return LOS2(X[1..n], null, false)
end function
function LOS2(X[1..n], prevValue, isEven)
    if n = 0 then
        return 0
    else
        if prevValue = null then
            result ← true
        else if isEven then
            result ← prevValue > X[1]
        else
            result ← prevValue < X[1]
        end if
        a ← LOS2(X[2..n], prevValue, isEven)
        if result then
            b ← 1+ LOS2(X[2..n], X[1], not isEven)
        else
            b ← 0
        end if
        return the maximum of a and b
    end if
end function
```

Longest Accelerating Subsequence

```
function LXS(X[1..n])
    return LXS2(X[1..n], null, null)
end function
function LXS2(X[1..n], prevValue, prevDiff)
    if n = 0 then
        return 0
    else
        if prevValue = null or prevDiff = null
then
            result ← true
        else

result ← X[1] − prevValue > prevDiff
        end if
        a ← LXS2(X[2..n], prevValue, prevDiff)
        if result then
            if prevValue = null then
                b ← 1+ LXS2(X[2..n], X[1], null)
            else
                b ← 1+
LXS2(X[2..n], X[1], X[1] − prevValue)
            end if
        else
            b ← 0
        end if
        return the maximum of a and b
    end if
end function
```

Subset Sum

Basic

```
function SubsetSum(X[1..n], T)
    if T = 0 then
        return True
    else if T < 0 or n = 0 then
        return False
    else
        return SubsetSum(X[1..n − 1], T) ∨
SubsetSum(X[1..n − 1], T − X[n])
    end if
end function
```

Memoized

```
function SubsetSum(X[1..n], T)
    S[n + 1, 0] ← True
    for t ← 1, T do
        S[n + 1, t] ← False
    end for
    for i ← n, 1 do
        S[i, 0] ← True
        for t ← 1, X[i] − 1 do
            S[i, t] ← S[i + 1, t] ▷ Avoid the case t < 0
        end for
        for t ← X[i], T do
            S[i, t] ← S[i + 1, t] ∨ S[i + 1, t − X[i]]
        end for
    end for
    return S[1, T]
end function
```

Dynamic Programming

1. Formulate the problem recursively
2. Build solutions to recurrence from the bottom up
   (a) Identify the subproblems
   (b) Analyze space and running time
   (c) Choose a data structure to memoize intermediate results
   (d) Identify dependencies between subproblems
   (e) Find a good evaluation order
   (f) Write down the algorithm

Longest Palindrome Subsequence

```
function LPS(text[1..n])
    memoized ← empty hash map
    memoized[the empty list] ← 0
    for i ← 1, n do
        for j ← 1, n − i do
            subproblem ← text[j..j + i]
            if i = 1 then
                memoized[subproblem] ← 1        ▷ A
single letter is always a palindrome of length one
            else
                r ← the maximum of
memoized[subproblem[2..i]] and
memoized[subproblem[1..i − 1]]
                if subproblem[1] = subproblem[i]
then
                    r ← the maximum of r and
2 + memoized[subproblem[2..i − 1]]
                end if
                memoized[subproblem] ← r
            end if
        end for
    end for
    return memoized[text]
end function
```