

micro

Classic Algorithms Cheat Sheet — derived from user-provided text

0/1 Knapsack (DP)

From provided text — concise explanation, complexity, and notes

Idea	Classic 0/1 knapsack using a 1D DP array (space optimized).
Time	See notes
Space	See notes

Key points

- Classic 0/1 knapsack using a 1D DP array (space optimized).
- Iterates items and fills dp from W down to wt[i] to avoid reuse of same item.
- Time: $O(nW)$, Space: $O(W)$.

Code (from source)

```
01 Knapsack
Int knapSack(int W, int wt[], int val[], int n){
    int i,w;
    int *dp = malloc((W+1)*sizeof(int));
    for(w=0; w<=W; ++w) dp[w]=0;
    for(i=0;i<n;i++){
        for(w=W; w>=wt[i]; --w){
            int take = dp[w-wt[i]] + val[i];
            if(take > dp[w]) dp[w] = take;
        }
    }
    int res = dp[W];
    free(dp);
    return res;
}
```

Matrix Chain Multiplication (DP)

From provided text — concise explanation, complexity, and notes

Idea	DP over chain lengths L from 2..n to compute minimal scalar multiplications.
Time	See notes
Space	See notes

Key points

- DP over chain lengths L from 2..n to compute minimal scalar multiplications.
- Time: $O(n^3)$, Space: $O(n^2)$.

Code (from source)

```
Matrix Chain Multiplication (DP)
int matrixChainOrder(int p[], int m){
    int n = m-1;
    int **dp = malloc((n+1)*sizeof(int*));
    for(int i=0;i<=n;i++){ dp[i]=malloc((n+1)*sizeof(int));
    memset(dp[i],0,(n+1)*sizeof(int)); }
    for(int L=2; L<=n; ++L){
        for(int i=1; i<=n-L+1; ++i){
            int j = i+L-1;
            dp[i][j] = INT_MAX;
            for(int k=i; k<j; ++k){
                int q = dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j];
                if(q < dp[i][j]) dp[i][j] = q;
            }
        }
    }
    int res = (n>=1) ? dp[1][n] : 0;
    for(int i=0;i<=n;i++) free(dp[i]);
    free(dp);
    return res;
}
```

Longest Common Subsequence (DP)

From provided text — concise explanation, complexity, and notes

Idea	Standard LCS dynamic programming with table reconstruction to get the sequence.
Time	See notes

Space	See notes
--------------	-----------

Key points

- Standard LCS dynamic programming with table reconstruction to get the sequence.
- Time: $O(mn)$, Space: $O(mn)$ (can be reduced if only length required).

Code (from source)

```

Longest Common Subsequence (DP)
int LCS(char *X, char *Y, char **out){
    int m = strlen(X), n = strlen(Y);
    int **L = malloc((m+1)*sizeof(int*));
    for(int i=0;i<=m;i++){ L[i]=calloc(n+1,sizeof(int)); }
    for(int i=1;i<=m;i++)
        for(int j=1;j<=n;j++)
            L[i][j] = (X[i-1]==Y[j-1]) ? L[i-1][j-1]+1 :
            ((L[i-1][j] > L[i][j-1]) ? L[i-1][j] : L[i][j-1]);
    int len = L[m][n];
    if(out){
        char *seq = malloc(len+1);
        seq[len]=0;
        int i=m, j=n, idx=len-1;
        while(i>0 && j>0){
            if(X[i-1]==Y[j-1]){ seq[idx--]=X[i-1]; i--; j--; }
            else if(L[i-1][j] > L[i][j-1]) i--;
            else j--;
        }
        *out = seq;
    }
    for(int i=0;i<=m;i++) free(L[i]);
    free(L);
    return len;
}

```

Floyd–Warshall (All-Pairs Shortest Paths)

From provided text — concise explanation, complexity, and notes

Idea	Triple nested loop to consider intermediate vertex k. Supports negative weights (no negative cycles).
Time	See notes
Space	See notes

Key points

- Triple nested loop to consider intermediate vertex k. Supports negative weights (no negative cycles).
- Time: $O(n^3)$, Space: $O(n^2)$.

Code (from source)

```
Floyd-Warshall (all-pairs shortest paths)
void floydWarshall(int n, int **dist){
    for(int k=0;k<n;k++)
        for(int i=0;i<n;i++) if(dist[i][k]!=INF)
            for(int j=0;j<n;j++) if(dist[k][j]!=INF){
                if(dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
}
```

Optimal Binary Search Tree (DP)

From provided text — concise explanation, complexity, and notes

Idea	DP computing minimal expected search cost given frequency of keys.
Time	See notes
Space	See notes

Key points

- DP computing minimal expected search cost given frequency of keys.
- Time: $O(n^3)$ naive due to summing and trying all roots; can be optimized to $O(n^2)$ with Knuth optimization if applicable.

Code (from source)

```
Optimal Binary Search Tree (DP)
int optimalBST(int freq[], int n){
    int **cost = malloc((n+2)*sizeof(int*));
    for(int i=0;i<=n;i++){ cost[i]=calloc(n+1,sizeof(int)); }
    for(int i=0;i<n;i++) cost[i][i] = freq[i];
    for(int L=2; L<=n; ++L){
        for(int i=0;i<=n-L;i++){
            cost[i][i+L] = cost[i][i+L-1] + freq[i+L];
            for(int j=i+1;j<=i+L-1;j++){
                cost[i][i+L] = min(cost[i][i+L], cost[i][j] + cost[j+1][i+L]);
            }
        }
    }
}
```

```

        int j = i+L-1;
        int sum = 0;
        for(int k=i;k<=j;k++) sum += freq[k];
        int best = INT_MAX;
        for(int r=i; r<=j; ++r){
            int left = (r>i) ? cost[i][r-1] : 0;
            int right = (r<j) ? cost[r+1][j] : 0;
            int c = left + right + sum;
            if(c < best) best = c;
        }
        cost[i][j] = best;
    }
}
int res = (n>0) ? cost[0][n-1] : 0;
for(int i=0;i<=n;i++) free(cost[i]);
free(cost);
return res;
}

```

Dijkstra ($O(n^2)$ adjacency matrix)

From provided text — concise explanation, complexity, and notes

Idea	Classic Dijkstra without a heap: $O(n^2)$ time, good for dense graphs or small n .
Time	See notes
Space	See notes

Key points

- Classic Dijkstra without a heap: $O(n^2)$ time, good for dense graphs or small n .
- Requires non-negative weights.

Code (from source)

```

Dijkstra (simple  $O(n^2)$  using adjacency matrix
void dijkstra(int n, int **graph, int src, int *dist){
    int *vis = calloc(n,sizeof(int));
    for(int i=0;i<n;i++) dist[i] = INF;
    dist[src]=0;
    for(int count=0; count<n-1; count++){
        int u=-1, best=INF;
        for(int v=0; v<n; v++) if(!vis[v] && dist[v] < best){ best =
        if(u== -1) break;
    }
}
```

```

    vis[u]=1;
    for(int v=0; v<n; v++){
        if(!vis[v] && graph[u][v] != INF && dist[u] != INF
            && dist[u] + graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
    }
}
free(vis);
}

```

Prim's MST ($O(n^2)$)

From provided text — concise explanation, complexity, and notes

Idea	Prim's algorithm using arrays (no heap) to get $O(n^2)$.
Time	See notes
Space	See notes

Key points

- Prim's algorithm using arrays (no heap) to get $O(n^2)$.
- Produces parent array defining MST; total weight computed at the end.

Code (from source)

```

Prim's MST
int primMST(int n, int **graph, int *parent){
    int *key = malloc(n*sizeof(int));
    int *inMST = calloc(n,sizeof(int));
    for(int i=0;i<n;i++){ key[i]=INT_MAX; parent[i]=-1; }
    key[0]=0;
    for(int count=0; count<n-1; count++){
        int u=-1, best=INT_MAX;
        for(int v=0; v<n; v++) if(!inMST[v] && key[v] < best)
{ best=key[v]; u=v; }
        if(u==-1) break;
        inMST[u]=1;
        for(int v=0; v<n; v++){
            if(graph[u][v] && !inMST[v] && graph[u][v] < key[v]){
                parent[v]=u; key[v]=graph[u][v];
            }
        }
    }
    int total=0;
    for(int i=1;i<n;i++) if(parent[i]!=-1) total += graph[i][parent[i]];
}
```

```

    free(key); free(inMST);
    return total;
}

```

Kruskal's MST (Union-Find)

From provided text — concise explanation, complexity, and notes

Idea	Sort edges by weight and union sets to avoid cycles.
Time	See notes
Space	See notes

Key points

- Sort edges by weight and union sets to avoid cycles.
- Time: $O(E \log E)$ dominated by sorting; union-find nearly $O(1)$ amortized per op.

Code (from source)

```

Kruskal's MST (edge list + union-find)
typedef struct { int u,v,w; } Edge;
int findp(int *p, int x){ return (p[x]==x)?x:(p[x]=findp(p,p[x])); }
int kruskalMST(Edge edges[], int E, int V){
    // sort edges by weight (simple qsort)
    int cmp(const void *a, const void *b)
    { return ((Edge*)a)->w - ((Edge*)b)->w; }
    qsort(edges, E, sizeof(Edge), cmp);
    int *parent = malloc(V*sizeof(int));
    int *rank = calloc(V,sizeof(int));
    for(int i=0;i<V;i++) parent[i]=i;
    int e=0, i=0, total=0;
    while(e < V-1 && i < E){
        Edge next = edges[i++];
        int x = findp(parent, next.u);
        int y = findp(parent, next.v);
        if(x!=y){
            // union by rank
            if(rank[x] < rank[y]) parent[x]=y;
            else if(rank[x] > rank[y]) parent[y]=x;
            else { parent[y]=x; rank[x]++; }
            total += next.w;
            e++;
        }
    }
}

```

```
    free(parent); free(rank);
    return (e==V-1) ? total : -1; // -1 if MST not found (disconnected)
}
```

Fractional Knapsack (Greedy)

From provided text — concise explanation, complexity, and notes

Idea	Sort items by value/weight ratio and take fractionally until capacity is full.
Time	See notes
Space	See notes

Key points

- Sort items by value/weight ratio and take fractionally until capacity is full.
- Greedy optimal. Time: $O(n \log n)$ due to sort.

Code (from source)

```
Fractional Knapsack (greedy)
typedef struct { double wt, val, ratio; } Item;
int cmpItem(const void *a, const void *b){
    double r1 = ((Item*)a)->ratio, r2 = ((Item*)b)->ratio;
    if(r1 < r2) return 1; if(r1 > r2) return -1; return 0;
}

double fractionalKnapsack(double W, int n, double wt[], double val[])
{
    Item *items = malloc(n*sizeof(Item));
    for(int i=0;i<n;i++){ items[i].wt = wt[i]; items[i].val = val[i];
        qsort(items, n, sizeof(Item), cmpItem);
    double remain = W, total=0.0;
    for(int i=0;i<n && remain>0;i++){
        if(items[i].wt <= remain){
            remain -= items[i].wt;
            total += items[i].val;
        } else {
            total += items[i].ratio * remain;
            remain = 0;
        }
    }
    free(items);
    return total;
}
```

Generated from user-provided source text. Each section is saved as a separate HTML file in /mnt/data/micro_cheatsheet/