Table of Contents

Dynamic Programming

To be able to use DP, the original problem must have:
1. Optimal sub-structure property:
    optimal solution to the problem contains within it optimal solutions to sub-problems
2. Overlapping sub-problems property
    we accidentally recalculate the same problem twice or more.

Matrix Chain Multiplication Problem (MCM)

Input: Matrices $A_1, A_2, ... A_n$, each A_i of size P_{i-1} x P_i

Output: Fully parenthesized product $A_1 A_2 ... A_n$ that minimizes the number of scalar multiplications

Step 2: Recursive formulation
Need to find $A_{1..n}$
Let m[i,j] = minimum number of scalar multiplications needed to compute $A_{i..j}$
Since $A_{i..j}$ can be obtained by breaking it into $A_{i..k}$ $A_{k+1..j}$, we have
m[i,j] = 0, if i=j
        = min i<=k<j { m[i,k]+m[k+1,j]+$p_{i-1} p_k p_j$ }, if i<j
let s[i,j] be the value k where the optimal split occurs.

Step 3 Computing the Optimal Costs

```
Matric-Chain-Order(p)
  n = length[p]-1
  for i = 1 to n do
    m[i,i] = 0
  for l = 2 to n do
    for i = 1 to n-l+1 do
      j = i+l-1
      m[i,j] = infinity
      for k = i to j-1 do
        q = m[i,k] + m[k+1,j] + pi-1*pk*pj
        if q < m[i,j] then
          m[i,j] = q
          s[i,j] = k
  return m and s
```

Step 4: Constructing an Optimal Solution

```
Print-MCM(s,i,j)
  if i=j then
    print Ai
  else
    print "(" + Print-MCM(s,1,s[i,j]) + "*" + Print-MCM(s,s[i,j]+1,j) +
")"
```

Longest Common Subsequence (LCS)

```
Input: Two sequence
Output: A longest common subsequence of those two sequences, see details
below.
/* change this constant if you want a longer subsequence */
#define MAX 100

char X[MAX],Y[MAX];
int i,j,m,n,c[MAX][MAX],b[MAX][MAX];

int LCSlength() {
  m=strlen(X);
  n=strlen(Y);

  for (i=1;i<=m;i++) c[i][0]=0;
  for (j=0;j<=n;j++) c[0][j]=0;

  for (i=1;i<=m;i++)
    for (j=1;j<=n;j++) {
      if (X[i-1]==Y[j-1]) {
        c[i][j]=c[i-1][j-1]+1;
        b[i][j]=1; /* from north west */
      }
      else if (c[i-1][j]>=c[i][j-1]) {
        c[i][j]=c[i-1][j];
        b[i][j]=2; /* from north */
      }
      else {
        c[i][j]=c[i][j-1];
        b[i][j]=3; /* from west */
      }
    }

  return c[m][n];
}

void printLCS(int i,int j) {
  if (i==0 || j==0) return;

  if (b[i][j]==1) {
    printLCS(i-1,j-1);
    printf("%c",X[i-1]);
  }
  else if (b[i][j]==2)
    printLCS(i-1,j);
  else
    printLCS(i,j-1);
}
```

Edit Distance (ED)

```
Input: Given two string, Cost for deletion, insertion, and replace
Output: Give the minimum actions needed to transform first string into
the second one.
Let d(string1,string2) be the distance between these 2 strings.
Recurrence Relation:
d("","") = 0
d(s ,"") = d("", s) = |s| ;; i.e. length of s
d(s1+ch1, s2+ch2)
  = min( d(s1, s2) + if (ch1==ch2) then 0 else 1,
         d(s1+ch1, s2) + 1,
         d(s1, s2+ch2) + 1 )
DP pseudo code:
A two-dimensional matrix, m[0..|s1|,0..|s2|] is used to hold the edit
distance values, such that m[i,j] = d(s1[1..i], s2[1..j]).

m[0][0] = 0;
for (i=1;i<length(s1);i++) m[i][0] = i;
for (j=1;j<length(s2);j++) m[0][j] = j;
for (i=0;i<length(s1);i++)
  for (j=0;j<length(s2);j++) {
    val = (s1[i] == s2[j]) ? 0 : 1;
    m[i][j] = min( m[i-1][j-1] + val,
                   min(m[i-1][j]+1 , m[i][j-1]+1));
  }
To output the trace, use another array to store our action along the way.
Trace back these values later.
```

Longest Inc/Decreasing Subsequence (LIS/LDS)

```
Input: Given a sequence
Output: The longest subsequence of the given sequence such that all
values in this longest  subsequence is strictly increasing/decreasing.
O(N^2) DP solution for LIS problem (this code check for increasing
values):
for i = 1 to total-1
  for j = i+1 to total
    if height[j] > height[i] then
      if length[i] + 1 > length[j] then
        length[j] = length[i] + 1
        predecessor[j] = i
```

Zero-One Knapsack (0-1)

Input: N items, each with various Vi (Value) and Wi (Weight) and max Knapsack size MW.
Output: Maximum value of items that one can carry, if he can either take or not-take a particular item.
Let C[i][w] be the maximum value if the available items are $\{X_1,X_2,...,X_i\}$ and the knapsack size is w.
Recurrence Relation:

```
;; if i == 0 or w == 0 (if no item or knapsack full), we can't take anything
C[i][w] = 0
;; if Wi > w (this item too heavy for our knapsack), skip this item
C[i][w] = C[i-1][w];
;; if Wi <= w, take the maximum of "not-take" or "take"
C[i][w] = max(C[i-1][w] , C[i-1][w-Wi]+Vi);
;; The solution can be found in C[N][W];
```

DP pseudo code:

```
for (i=0;i<=N ;i++) C[i][0] = 0;
for (w=0;w<=MW;w++) C[0][w] = 0;

for (i=1;i<=N;i++)
  for (w=1;w<=MW;w++) {
    if (Wi[i] > w)
      C[i][w] = C[i-1][w];
    else
      C[i][w] = max(C[i-1][w] , C[i-1][w-Wi[i]]+Vi[i]);
  }

output(C[N][MW]);
```

Note: actually, top-down is faster than bottom up in this problem since we unnecessarily compute too much thing if MW is big.

Counting Change (CC)

Input: A list of denominations and a value N to be changed with these denominations
Output: Number of ways to change N
The number of ways to change amount A using N kinds of coins equals to:

1. The number of ways to change amount A using all but the first kind of coins, +
2. The number of ways to change amount A-D using all N kinds of coins, where D is the denomination of the first kind of coin.

The tree recursive process will gradually reduce the value of A, then using this rule, we can determine how many ways to change coins.

1. If A is exactly 0, we should count that as 1 way to make change.
2. If A is less than 0, we should count that as 0 ways to make change.
3. If N kinds of coins is 0, we should count that as 0 ways to make change.

```
#define MAXTOTAL 10000

long long nway[MAXTOTAL+1];
/* Assume we have 5 different coins here */
int coin[5]={ 50,25,10,5,1 };

void main() {
  int i,j,n,v, c;
  scanf("%d",&n);
  v=5;
  nway[0]=1;
  for (i=0;i<v;i++) {
    c=coin[i];
    for (j=c;j<=n; j++)
      nway[j]+=nway[j-c];
  }
  printf("%lld\n",nway[n]);
}
```

Graph Algorithms

Topological Sort

Given a collection of objects, along with some ordering constraints, such as "A must be before B," find an order of the objects such that all the ordering constraints hold.
Algorithm: Create a directed graph over the objects, where there is an arc from A to B if "A must be before B." Make a pass through the objects in arbitrary order. Each time you find an object with in-degree of 0, greedily place it on the end of the current ordering, delete all of its out-arcs, and recurse on its (former) children, performing the same check. If this algorithm gets through all the objects without putting every object in the ordering, there is no ordering which satisfies the constraints.

BFS

```
Busca_em_largura(int G[MAX_NOS][MAX_NOS], int n, int no_inicial) {
  int i, no_atual;
  int fila[MAX_NOS], ini, fim;
  ini=fim=0;
  fila[fim++]=no_inicial;
  while(ini!=fim) {
    no_atual=fila[ini++];
    Visita(no_atual);
    for(i=0; i<n; i++)
       if(G[no_atual][i]!=INF)
          fila[fim++]=i;
  }
}
```

BFS runs in O(V+E)
Note: BFS can compute d[v] = shortest-path distance from s to v, in terms of minimum number of edges from s to v (un-weighted graph). Its breadth-first tree can be used to represent the shortest-path.

Dijkstra

```
int dijkstra(int origem, int destino, int n) {
 int i,minimo,atual;
 int pred[NMAX], passou[NMAX], custo[NMAX];
 for(i=0;i<n;i++) {
    pred[i] = -1;
    passou[i] = 0;
    custo[i] = INF;
  }
  custo[origem] = 0;
  atual = origem;
  while(atual != destino) {
    for(i=0;i<n;i++)
      if (grafo[atual][i] != -1)
        if (custo[atual] + grafo[atual][i] < custo[i]) {
          custo[i] = custo[atual] + grafo[atual][i];
          pred[i] = atual;
        }
    minimo = INF + 1;
    passou[atual] = 1;
    for(i=0;i<n;i++)
      if((custo[i] < minimo) && (!passou[i])) {
        minimo = custo[i];
        atual = i;
      }
    if(minimo >= INF)
      return INF;
  }
  return custo[destino];
}
```

Bellman-Ford Algorithm

A more generalized single-source shortest paths algorithm which can find the shortest path in a graph with negative weighted edges. If there is no negative cycle in the graph, this algorithm will updates each d[v] with the shortest path from s to v, fill up the predecessor list "pi", and return TRUE. However, if there is a negative cycle in the given graph, this algorithm will return FALSE.

```
BELLMAN_FORD(Graph G,double w[][],Node s)
  initialize_single_source(G,s)
  for i=1 to |V[G]|-1
    for each edge (u,v) in E[G]
      relax(u,v,w)

  for each edge (u,v) in E[G]
    if d[v] > d[u] + w(u, v) then
      return FALSE
return TRUE
```

Floyd-Warshall

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++) {
    d[i][j] = w[i][j];
    p[i][j] = i;
  }

for (i=0; i<n; i++)
  d[i][i] = 0;
for (k=0;k<n;k++) /* k -> is the intermediate point */
  for (i=0;i<n;i++) /* start from i */
    for (j=0;j<n;j++) /* reaching j */
      /* if i-->k + k-->j is smaller than the original i-->j */
      if (d[i][k] + d[k][j] < d[i][j]) {
        /* then reduce i-->j distance to the smaller one i->k->j */
        graph[x][y] = graph[x][k]+graph[k][y];
        /* and update the predecessor matrix */
        p[i][j] = p[k][j];
      }
```

In the k-th iteration of the outer loop, we try to improve the currently known shortest paths by considering k as an intermediate node. Therefore, after the k-th iteration we know those shortest paths that only contain intermediate nodes from the set {0, 1, 2,...,k}. After all n iterations we know the real shortest paths.

```
print_path (int i, int j) {
  if (i!=j)
    print_path(i,p[i][j]);
  print(j);
}
```

Strongly Connected Components

Input: A directed graph G = (V,E)
Output: All strongly connected components of G, where in strongly connected component, all pair of vertices u and v in that component, we have u ~~> v and v ~~> u, i.e. u and v are reachable from each other.

```
Strongly-Connected-Components(G)
1. call DFS(G) to compute finishing times f[u] for each vertex u ~~ O(V+E)
2. compute GT, inversing all edges in G ~~ O(V+E) using adjacency list
3. call DFS(GT), but in the main loop of DFS, consider the vertices in order of decreasing
    f[u] as computed in step 1 ~~ O(V+E)
4. output the vertices of each tree in the depth-first forest of step 3 as a separate
    strongly connected component
Strongly-Connected-Components runs in O(3(V+E)) ~~ O(V+E)
```

Edmonds-Karp Maximum Network Flow

```
#include <vector>
#include <queue>
#include <iostream>
using namespace std;
/* Gets the adjacency matrix, returns a matrix with each edge's flow */
vector<vector<int> > max_flow(vector<vector<int> > M, int source, int
sink) {
  int sentry = 2000000000;
  vector<int> tl(M[0].size(),0);
  vector<vector<int> > flow(M.size(),tl);
  bool finished = false;
  while (!finished) {
     vector<int> P(M.size(),-1);
    queue<int> Q;
    Q.push(source);
    P[source] = source;
    int curr;
    while (Q.size() > 0) {
      curr = Q.front();
      Q.pop();
      for (int i = 0; i < M[curr].size(); ++i) {
        if (M[curr][i] != 0 && P[i] == -1) {
          Q.push(i);
          P[i] = curr;
        }
      }
    }
    curr = sink;
    int maxFlow = sentry;
    int next;
    while (P[curr] != -1 && P[curr] != curr) {
      next = P[curr];
      maxFlow = min(M[next][curr],maxFlow);
      curr = next;
    }
    if (maxFlow == 0 || maxFlow == sentry) {
      finished = true;
    } else {
      finished = false;
      curr = sink;
      while (P[curr] != -1 && P[curr] != curr) {
        next = P[curr];
        M[curr][next] += maxFlow;
        M[next][curr] -= maxFlow;
        flow[curr][next] -= maxFlow;
        flow[next][curr] += maxFlow;
        curr = next;
      }
    }
  }
  return flow;
}
```

Geometric Algorithms

Lines

```
typedef struct {
        double a;               /* x-coefficient */
        double b;               /* y-coefficient */
        double c;               /* constant term */
} line;
points_to_line(point p1, point p2, line *l) {
        if (p1[X] == p2[X]) {
                l->a = 1;
                l->b = 0;
                l->c = -p1[X];
        } else {
                l->b = 1;
                l->a = -(p1[Y]-p2[Y])/(p1[X]-p2[X]);
                l->c = -(l->a * p1[X]) - (l->b * p1[Y]);
        }
}
point_and_slope_to_line(point p, double m, line *l) {
        l->a = -m;
        l->b = 1;
        l->c = -((l->a*p[X]) + (l->b*p[Y]));
}
```

Line Intersection

```
bool parallelQ(line l1, line l2) {
     return ( (fabs(l1.a-l2.a) <= EPSILON) &&
              (fabs(l1.b-l2.b) <= EPSILON) );
}
intersection_point(line l1, line l2, point p) {
     if (same_lineQ(l1,l2)) {
         printf("Warning: Identical lines, all points intersect.\n");
         p[X] = p[Y] = 0.0;
         return;
     }

     if (parallelQ(l1,l2) == TRUE) {
         printf("Error: Distinct parallel lines do not intersect.\n");
         return;
     }

     p[X] = (l2.b*l1.c - l1.b*l2.c) / (l2.a*l1.b - l1.a*l2.b);

     if (fabs(l1.b) > EPSILON)       /* test for vertical line */
             p[Y] = - (l1.a * (p[X]) + l1.c) / l1.b;
     else
             p[Y] = - (l2.a * (p[X]) + l2.c) / l2.b;
}
```

Angle Between Two Lines

$$\tan\theta = \frac{a_1 b_2 - a_2 b_1}{a_1 a_2 + b_1 b_2}$$

Closest Point

```
closest_point(point p_in, line l, point p_c) {
      line perp;             /* perpendicular to l through (x,y) */
      if (fabs(l.b) <= EPSILON) {     /* vertical line */
              p_c[X] = -(l.c);
              p_c[Y] = p_in[Y];
              return;
      }
      if (fabs(l.a) <= EPSILON) {     /* horizontal line */
              p_c[X] = p_in[X];
              p_c[Y] = -(l.c);
              return;
      }
      point_and_slope_to_line(p_in,1/l.a,&perp); /* normal case */
      intersection_point(l,perp,p_c);
}
```

Testing Intersection

```
typedef struct {
        point p1,p2;            /* endpoints of line segment */
} segment;

bool segments_intersect(segment s1, segment s2) {
   line l1,l2;      /* lines containing the input segments */
   point p;         /* intersection point */
   points_to_line(s1.p1,s1.p2,&l1);
   points_to_line(s2.p1,s2.p2,&l2);
   if (same_lineQ(l1,l2))  /* overlapping or disjoint segments */
           return( point_in_box(s1.p1,s2.p1,s2.p2) ||
                   point_in_box(s1.p2,s2.p1,s2.p2) ||
                   point_in_box(s2.p1,s1.p1,s1.p2) ||
                   point_in_box(s2.p1,s1.p1,s1.p2) );
   if (parallelQ(l1,l2)) return(FALSE);
   intersection_point(l1,l2,p);
   return(point_in_box(p,s1.p1,s1.p2) && point_in_box(p,s2.p1,s2.p2));
}
bool point_in_box(point p, point b1, point b2) {
      return( (p[X] >= min(b1[X],b2[X])) && (p[X] <= max(b1[X],b2[X]))
           && (p[Y] >= min(b1[Y],b2[Y])) && (p[Y] <= max(b1[Y],b2[Y])) );
}
```

Triangle Orientation

```
bool ccw(point a, point b, point c) {
      double signed_triangle_area();
      return (signed_triangle_area(a,b,c) > EPSILON);
}
bool cw(point a, point b, point c) {
      double signed_triangle_area();
      return (signed_triangle_area(a,b,c) < EPSILON);
}
bool collinear(point a, point b, point c) {
      double signed_triangle_area();
      return (fabs(signed_triangle_area(a,b,c)) <= EPSILON);
}
```

Convex Hull

```
typedef struct {
  int x, y;
} ponto_t;

ponto_t origem;

int produtoVetorial(a, b, c)
     ponto_t a, b, c; {
  ponto_t p1, p2;
  p1.x = b.x - a.x;
  p1.y = b.y - a.y;
  p2.x = c.x - a.x;
  p2.y = c.y - a.y;
  return p1.x*p2.y - p2.x*p1.y;
}

int dist2(a, b)
     ponto_t a, b; {
  return (a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y);
}

int compara(void *a, void *b) {
  ponto_t c, d;
  int e;
  c = *(ponto_t *)a;
  d = *(ponto_t *)b;
  e = produtoVetorial(origem,c,d);
  if(e == 0) {
    if(dist2(origem,c) > dist2(origem,d))
      return 1;
    return -1;
  }
  if (e < 0)
    return 1;
  return -1;
}
```

```
/*assume que o poligono estah em p[] e poe o convexHull em q[]*/
/*retorna o numero de pontos em q[]*/
int convexHull(int n) {
      m = 0; /*escolhe origem*/
      for(i=1;i<n;i++)
       if(p[i].y < p[m].y ||
         (p[i].y == p[m].y && p[i].x < p[m].x))
        m = i;
      aux = p[0];
      p[0] = p[m];
      p[m] = aux;
      origem = p[0]; /*fim de escolhe origem*/
      qsort(p+1,n-1,sizeof(ponto_t),compara);
      for(i=0;i<n;i++) /*elimina colineares*/
        v[i] = 1;
      for(i=1;i<n-1;i++)
        if(produtoVetorial(p[i-1],p[i],p[i+1])==0)
          v[i] = 0;
      j = 0;
      for(i=0;i<n;i++)
        if(v[i])
          q[j++] = p[i];
      n = j;
      for(i=0;i<n;i++)
        p[i] = q[i]; /*fim de elimina colineares*/
      topo = 0; /*inicializa solucao do convexHull*/
      for(i=0;i<3;i++)
        q[topo++] = p[i];
      for(i=3;i<n;i++) { /*graham-scan*/
        while(produtoVetorial(q[topo-2],q[topo-1],p[i]) < 0)
          topo--;
        q[topo++] = p[i];
      }
      return topo;
}
```

Polygon Area

```
double area(polygon *p) {
        double total = 0.0;             /* total area so far */
        int i, j;                       /* counters */

        for (i=0; i<p->n; i++) {
             j = (i+1) % p->n;
             total += (p->p[i][X]*p->p[j][Y]) - (p->p[j][X]*p->p[i][Y]);
        }
        return(total / 2.0);
}
```

Point Inside a Polygon

```
struct point { int x, y; char c; };
struct line { struct point p1, p2; };
struct point polygon[Nmax];

int ccw(struct point p0,
        struct point p1,
        struct point p2 )
  {
    int dx1, dx2, dy1, dy2;
    dx1 = p1.x - p0.x; dy1 = p1.y - p0.y;
    dx2 = p2.x - p0.x; dy2 = p2.y - p0.y;
    if (dx1*dy2 > dy1*dx2) return +1;
    if (dx1*dy2 < dy1*dx2) return -1;
    if ((dx1*dx2 < 0) || (dy1*dy2 < 0)) return -1;
    if ((dx1*dx1+dy1*dy1) < (dx2*dx2+dy2*dy2))
                                    return +1;
    return 0;
  }

int intersect(struct line l1, struct line l2)
  {
    return ((ccw(l1.p1, l1.p2, l2.p1)
            *ccw(l1.p1, l1.p2, l2.p2)) <= 0)
        && ((ccw(l2.p1, l2.p2, l1.p1)
            *ccw(l2.p1, l2.p2, l1.p2)) <= 0);
  }

int inside(struct point t, struct point p[], int N)
  {
    int i, count = 0, j = 0;
    struct line lt,lp;
    p[0] = p[N]; p[N+1] = p[1];
    lt.p1 = t; lt.p2 = t; lt.p2.x = INT_MAX;
    for (i = 1; i <= N; i++)
      {
        lp.p1= p[i]; lp.p2 = p[i];
        if (!intersect(lp,lt))
          {
            lp.p2 = p[j]; j = i;
            if (intersect(lp,lt)) count++;
          }
      }
    return count & 1;
  }
```

Numeric Algorithms

Máximo divisor comum estendido

Entrada:
a, b, Números naturais
Saída:
retorno, mdc entre *a* e *b.*
xx e *yy* tais que *mdc(a,b) = xx.a + yy.b*
Complexidade: *O(log(min(a,b))*

```
int mdc(int a, int b, int *x, int *y)
{
  int ret, xx, yy;
  if(a<0) a=-a;
  if(b<0) b=-b;
  if(b==0) {
    *x=1; *y=0;
    return a;
  }
  ret=mdc(b, a%b, &xx, &yy);
  *x = yy;
  *y = xx – a/b*yy;
  return ret;
}
```

Exponenciação rápida

Entrada: inteiros *a*, *b* e *n*
Saída: a^b mod *n*
Complexidade: ???

```
int modexp(int a, int b, int n)
{
  long long res;
  if(b==0)
    return 1;
  else {
    res=modexp(a, b/2, n);
    res=(res*res)%n;
    if(b%2==1)
      res=(res*a)%n;
    return (int) res;
  }
}
```

Números Primos

Entrada: inteiro *n*
Saída:
primo[], números primos £ *n*
np, quantidade de números primos £ *n*
Complexidade: *O(n log n)*

```
void AchaPrimos(int n, int *primo)
{
  int i, j;
  primo[1]=0; primo[2]=1;
  for(i=3; i<=n; i++)
    primo[i]=i%2;
  for(i=3; i*i<=n; i+=2)
    if(primo[i])
      for(j=i*i; j<=n; j+=i)
        primo[j]=0;
  np=0;
  for(i=1; i<=n; i++)
    if(primo[i])
      primo[np++]=i;
}
```