データ構造とアルゴリズム実験レポート

＜課題 8：動的計画法＞

202213025 – 3 クラス – Can Minh Nghia

締切日：2024/2/5

提出日：2024/2/5

必須課題

注意：Makefile の内容は課題によって違う。

課題 8-1

8-1-1 実装の仕様

動的計画法を用いずに，下の式を直接用いて再帰的にナップサック問題の最適解を探索する関数 knapsack を実装しなさい． なお，選択された荷物の集合は計算しなくて良い．

G_{k,i}は k 番目までの荷物に対して, ナップサックの容量を i としたときの最適解である． G が式の右辺に現れるので再帰的な関数になる．

$$G_{k,i} = \begin{cases} G_{k-1,i} & \text{if } i - w_k < 0 \\ \max(G_{k-1,i}, G_{k-1,i-w_k} + v_k) & \text{otherwise} \end{cases}$$

以下の要件を確認すること．

教科書の表 6.1 の例に関して，荷物の数が 4, 容量が 3,4,5 の場合の実行結果が正しいことを確認すること.

knapsack 関数の実行時間が荷物の数に関して指数関数的であることを確認せよ．上のプログラムの引数として荷物の数を与え，荷物の数が 25〜30 の場合の実行時間を確認すれば良い.

8-1-2 実装コードおよび実装コードの説明

Makefile: knapsack: knapsack.o

knapsack.c ファイルの **knapsack()関数**を解説する。

```
14 int knapsack(int v[], int w[], int k, int i) {
15   if (k == 0 || i == 0) return 0;
16
17     if (w[k] > i) // If weight of that product is bigger than knapsack
18         return knapsack(v, w, k-1, i); // skip that product
19
20     // Choosing or not choosing the current product, depends on which one has more value
21     return max(v[k] + knapsack(v, w, k - 1, i - w[k]), knapsack(v, w, k-1, i));
22
23 }
```

Line 15: 荷物なしまたはナップサックなしの場合、0 を返す。

Line 17~18: 式の上の部分に相当する。

$$G_{k-1,i} \qquad \text{if } i - w_k < 0$$

つまり、現在の荷物がナップザックより重かったら、前の荷物に戻る。

Line 21: 式の下の部分に相当する。

$$\max(G_{k-1,i}, G_{k-1,i-w_k} + v_k) \quad \text{otherwise}$$

つまり、現在の荷物を選ぶか選ばないか、どれの価値が高いのかによって決める。

knapsack.c ファイルの **main()関数**はサンプルコードの通りである。しかし、以下のように 4＜i を 5＜i に変更する。なぜかというと、表 6.1 のナップザックの容量は 5 だからだ。

```
31  // 引数処理
32  if (argc == 3) {
33    int k = atoi(argv[1]);
34    int i = atoi(argv[2]);
35    if (k < 1 || 4 < k || i < 1 || 5 < i) {
36      fprintf(stderr, "k と i は [1, 5] の範囲で指定してください。");
37      return 1;
38    }
```

8-1-3 実行結果：

教科書の表 6.1 の例に関して，荷物の数が 4, 容量が 3,4,5 の場合の実行結果が正しいことを確認すること.

表 6.1　荷物の価値と重さ

荷物	価値（円）	重さ (kg)
1	250	1
2	380	2
3	420	4
4	520	3

		容量 (kg)				
		1	2	3	4	5
荷物	1	250,{1}	250,{1}	250,{1}	250,{1}	250,{1}
	2	250,{1}	380,{2}	630,{1,2}	630,{1,2}	630,{1,2}
	3	250,{1}	380,{2}	630,{1,2}	630,{1,2}	670,{1,3}
	4	250,{1}	380,{2}	630,{1,2}	770,{1,4}	900,{2,4}

図 6.5　動的計画法による処理の結果（ナップザック問題）

容量=3

```
azalea02:~ s2213025$ ./knapsack 4 3
結果：630
```

容量=4

```
azalea02:~ s2213025$ ./knapsack 4 4
結果：770
```

容量=5

```
azalea02:~ s2213025$ ./knapsack 4 5
結果：900
```

教科書の図 6.5 と比べると、あっていることがわかる。

knapsack 関数の実行時間が荷物の数に関して指数関数的であることを確認せよ．上のプログラムの引数として荷物の数を与え，荷物の数が 25〜30 の場合の実行時間を確認すれば良い．

荷物の数=1

```
azalea02:~ s2213025$ ./knapsack 1
容量：5
結果：47
実行時間：0.000000 sec.
```

荷物の数=5

```
azalea02:~ s2213025$ ./knapsack 5
容量：25
結果：175
実行時間：0.000002 sec.
```

荷物の数=10

```
azalea02:~ s2213025$ ./knapsack 10
容量：50
結果：534
実行時間：0.000171 sec.
```

荷物の数=15

```
azalea02:~ s2213025$ ./knapsack 15
容量：75
結果：845
実行時間：0.040329 sec.
```

荷物の数=20

```
azalea02:~ s2213025$ ./knapsack 20
容量：100
結果：969
実行時間：9.676928 sec.
```

荷物の数=21

```
azalea02:~ s2213025$ ./knapsack 21
容量：105
結果：956
実行時間：28.917430 sec.
```

よって、knapsack 関数の実行時間が荷物の数に関して指数関数的である

課題 8-2

8-2-1 実装の仕様

動的計画法を用いてナップサック問題の最適解を探索するプログラム knapsackDP.c を実装しなさい．教科書 p.132, リスト 6.3 を参照するとよい．なお，選択された荷物の集合は計算しなくて良い．

以下の要件を確認すること．

教科書の表 6.1 の例に関して，荷物の数が 4, 容量が 3,4,5 の場合の実行結果が正しいことを確認すること．

動的計画法を用いない実装よりも遥かに大きな問題を解けることを確認せよ．動的計画法による実装の時間計算量に関して，実験により議論した場合は加点する．

8-2-2 実装コードおよび実装コードの説明

Makefile: knapsackDP: knapsackDP.o

knapsackDP.c ファイルの **knapsack()関数**を解説する。

```
13 int knapsack(int v[], int w[], int k, int i) { //i is volume of knapsack, k is products
14   int G[k+1][i+1]; //G is the knapsack value array
15   //int S[k][i]; //S is the tracking array
16
17   if (k == 0 || i == 0) return 0;
18   for (int a = 0; a <= k; a++) {
19       for (int b = 0; b <= i; b++) {
20             G[a][b] = 0;
21             //S[a][b] = -1;
22        }
23  }
```

Line 17: 荷物なしまたはナップサックなしの場合、0 を返す。

Line 18~23: ナップサック行列の作成。

```
25        //fill the knapsack
26        for (int a = 1; a <= k; a++) {
27              for (int b = 1; b <= i; b++) {
28                    if (b < w[a]) {
29                          G[a][b] = G[a-1][b];
30
31                    } else {
32                          if (G[a-1][b] < G[a-1][b-w[a]] + v[a]) {
33                                G[a][b] = G[a-1][b-w[a]] + v[a];
34
35                          } else {
36                                G[a][b] = G[a-1][b];
37
38                          }
39                    }
40              }
41        }
42
43        for (int a = 0; a <= k; a++) {
44              for (int b = 0; b <= i; b++) {
45                    printf("%d ", G[a][b]);
46              }
47              printf("\n");
48        }
49
50        return G[k][i];
51  }
```

Line 26~29: 現在の荷物が現在のナップサックの容量より重かったら、前の荷物の重さにする。

Line 31~36: そうでなければ、現在の荷物を選ぶか選ばないか処理を行う。Line 32~33 は現在の荷物を選ぶための処理で、line 35~36 は荷物を選ばないための処理である。選ぶか選ばないかは基本課題 1 で与えられた式によって決める。ただし、k の代わりに a を、i の代わりに b を使ってプログラミングをする。

$$G_{k,i} = \begin{cases} G_{k-1,i} & \text{if } i - w_k < 0 \\ \max(G_{k-1,i}, G_{k-1,i-w_k} + v_k) & \text{otherwise} \end{cases}$$

Line 43~48:デバッグのためにナップサックを printf 関数で表す。

Line 50: 行列の下右の角にある値は最適解であるから、それを返す。

8-2-3 実行結果：

教科書の表 6.1 の例に関して，荷物の数が 4, 容量が 3,4,5 の場合の実行結果が正しいことを確認すること.

表 6.1　荷物の価値と重さ

荷物	価値（円）	重さ (kg)
1	250	1
2	380	2
3	420	4
4	520	3

		容量 (kg)				
		1	2	3	4	5
荷物	1	250,{1}	250,{1}	250,{1}	250,{1}	250,{1}
	2	250,{1}	380,{2}	630,{1,2}	630,{1,2}	630,{1,2}
	3	250,{1}	380,{2}	630,{1,2}	630,{1,2}	670,{1,3}
	4	250,{1}	380,{2}	630,{1,2}	770,{1,4}	900,{2,4}

図 6.5　動的計画法による処理の結果（ナップザック問題）

容量=3

```
azalea02:~ s2213025$ ./knapsackDP 4 3
0 0 0 0
0 250 250 250
0 250 380 630
0 250 380 630
0 250 380 630
結果：630
```

容量=4

```
azalea02:~ s2213025$ ./knapsackDP 4 4
0 0 0 0 0
0 250 250 250 250
0 250 380 630 630
0 250 380 630 630
0 250 380 630 770
結果：770
```

容量=5

```
azalea02:~ s2213025$ ./knapsackDP 4 5
0 0 0 0 0 0
0 250 250 250 250 250
0 250 380 630 630 630
0 250 380 630 630 670
0 250 380 630 770 900
結果：900
```

よって、あっていることがわかる。

動的計画法を用いない実装よりも遥かに大きな問題を解けることを確認せよ.

荷物の数=1

```
azalea02:~ s2213025$ ./knapsackDP 1
容量：5
結果：0
実行時間：0.000001 sec.
```

荷物の数=5

```
azalea02:~ s2213025$ ./knapsackDP 5
容量：25
結果：219
実行時間：0.000001 sec.
```

荷物の数=10

```
azalea02:~ s2213025$ ./knapsackDP 10
容量：50
結果：477
実行時間：0.000003 sec.
```

荷物の数=15

```
azalea02:~ s2213025$ ./knapsackDP 15
容量：75
結果：755
実行時間：0.000005 sec.
```

荷物の数=20

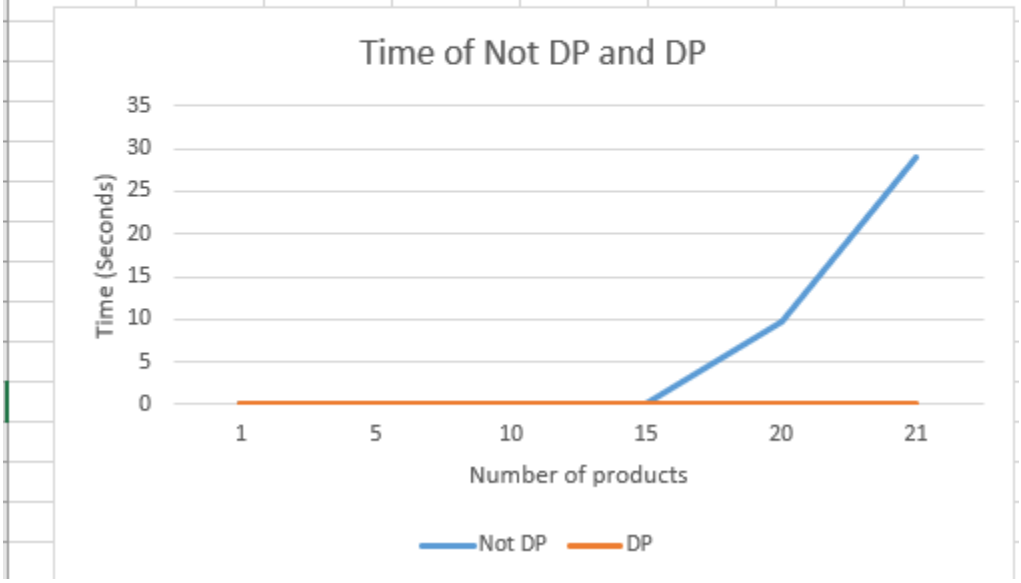
```
azalea02:~ s2213025$ ./knapsackDP 20
容量：100
結果：948
実行時間：0.000010 sec.
```

荷物の数=21

```
azalea02:~ s2213025$ ./knapsackDP 21
容量：105
結果：952
実行時間：0.000011 sec.
```

動的計画法による実装の時間計算量に関して，実験により議論した場合は加点する．

	1	5	10	15	20	21
Not DP	0	0.000002	0.000171	0.040329	9.676928	28.91743
DP	0.000001	0.000001	0.000003	0.000005	0.00001	0.000011



上の図は動的計画法を利用しない実装(Not DP)と動的計画法を利用する実装(DP)の計算時間を表すものである。図から明らかに見えるとおりに、荷物の数が 15 個までに処理時間がほとんど同じだけど、15 個からは Not DP の方が急激に増加したことがわかる。理由として、Not DP が再帰法を使うから、荷物の数が多ければ多いほど計算量がその分非常に増加すると判断する。一方で、動的計画法は行列を使うから、どうせ検索しなければならないのは行列の要素だけだから、計算量が Not DP に比べて非常に少ないといえる。

考察と感想

動的計画法は少しわかりにくかったが、今回のアルゴリズムが教科書に書いてあるから、そんなに困らなかった。ただし、基本課題 1 には荷物の数が 25〜30 の場合を確認すべきだと書いてあるが、ターミナルで荷物の数が 22 個以上だったらもう実行できない（どれぐらい待っても実行が終わらない）ので最大限を 21 個にした。

発展課題

課題 8-1

Makefile: knapsackDP2: knapsackDP2.o

8-1-1 実装の仕様

動的計画法によるナップサック問題のプログラムを選択された荷物を表示するように拡張した knapsackDP2.c を実装しなさい． 真偽値型の 2 次元配列 S [n+1][C+1] を用いて選択された荷物の集合を計算すると良い．最適解 G[k][i] を得るために k 番目の荷物を選択した場合に S[k][i]を真にし，それ以外の場合は偽とする．k-1 番目までの荷物が選択されたかは，最適解 G[k][i] をどう得たかを考えれば分かる．

knapsack 関数は, 配列 G[][]と配列 S[][]から選択された荷物の集合を表す配列 SS[]を計算し，配列 S[]を結果として返す.

8-1-2 実装コードおよび実装コードの説明

knapsack()関数だけ説明する。他にはサンプルコードの knapsackDP2.c と同じなので説明しない。

```
40 bool* knapsack(int v[], int w[], int n, int C) {
41     int** G = makeIntMatrix(n+1, C+1);
42     bool** S = makeBoolMatrix(n, C);
43     bool* SS = (bool*)malloc(sizeof(bool) * (n + 1));
44 
45   if (n == 0 || C == 0) return 0;
46   for (int a = 0; a <= n; a++) {
47       for (int b = 0; b <= C; b++) {
48             G[a][b] = 0;
49             S[a][b] = false;
50 
51         }
52   }
```

```
54      //fill the knapsack
55      for (int a = 1; a <= n; a++) {
56            for (int b = 1; b <= C; b++) {
57                  if (b < w[a]) {
58                        G[a][b] = G[a-1][b];
59                        S[a][b] = false;
60
61                  } else {
62                        if (G[a-1][b] < G[a-1][b-w[a]] + v[a]) {
63                              G[a][b] = G[a-1][b-w[a]] + v[a];
64                              S[a][b] = true;
65
66                        } else {
67                              G[a][b] = G[a-1][b];
68                              S[a][b] = false;
69
70                        }
71                  }
72            }
73      }
74
75      for (int a = 0; a <= n; a++) {
76            for (int b = 0; b <= C; b++) {
77                  printf("%d ", G[a][b]);
78            }
79            printf("\n");
80      }
```

これは基本課題 2 のソースコードである。基本課題 2 で説明したからもう余計な説明はしない。ただし、加えたのは Line 49, 59, 64 と 68 である。S[a][b]=true の場合は基本課題 1 に与えられた式のこの部分だけで、

$$\max(G_{k-1,i}, G_{k-1,i-w_k} + v_k) \quad \text{otherwise}$$

なぜかというと、これまで持っている荷物に加えて他の荷物を持つように

なって始めた/これまで持っている荷物(の一部)を捨てて他の荷物を持つようになって始めた時点だからだ。他の場合は S[a][a]=false。

8-1-3 実行結果：

教科書の表 6.1 の例に関して，荷物の数が 4, 容量が 3,4,5 の場合の実行結果が正しいことを確認すること.

容量=5

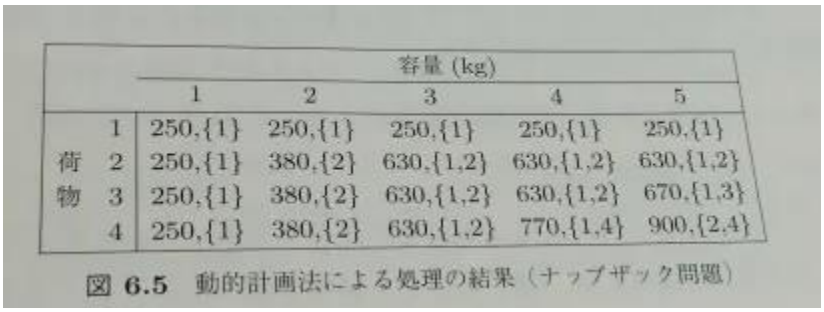
```
azalea02:~ s2213025$ ./knapsackDP2 4 5
0 0 0 0 0 0
0 250 250 250 250 250
0 250 380 630 630 630
0 250 380 630 630 670
0 250 380 630 770 900
重さ 2 価値 380
重さ 3 価値 520
合計価値 900
```

容量=4

```
azalea02:~ s2213025$ ./knapsackDP2 4 4
0 0 0 0 0
0 250 250 250 250
0 250 380 630 630
0 250 380 630 630
0 250 380 630 770
重さ 1 価値 250
重さ 3 価値 520
合計価値 770
```

容量=3

```
azalea02:~ s2213025$ ./knapsackDP2 4 3
0 0 0 0
0 250 250 250
0 250 380 630
0 250 380 630
0 250 380 630
重さ 1 価値 250
重さ 2 価値 380
合計価値 630
```



		容量 (kg)				
		1	2	3	4	5
荷物	1	250,{1}	250,{1}	250,{1}	250,{1}	250,{1}
	2	250,{1}	380,{2}	630,{1,2}	630,{1,2}	630,{1,2}
	3	250,{1}	380,{2}	630,{1,2}	630,{1,2}	670,{1,3}
	4	250,{1}	380,{2}	630,{1,2}	770,{1,4}	900,{2,4}

図 6.5 動的計画法による処理の結果（ナップザック問題）

教科書の図 6.5 と同じだから、あっていることがわかる。

配列 v[], w[]を変更し荷物の個数 10 の場合について実行結果を確認せよ．幾つかのナップサックの容量に関して確認すること．

main()関数の Line 100, 101 はこのように加える：

```
 95 int main(int argc, char** argv) {
 96   /* 教科書：表 6.1の例
 97       v[1]～v[4]：価格
 98       w[1]～w[4]：重さ */
 99   int num = 4;
100   int v[] = {0, 250, 380, 420, 520, 453, 564, 677, 423, 876, 421};
101   int w[] = {0, 1, 2, 4, 3, 5, 6, 7, 8, 9, 10 };
```

容量=5

```
azalea02:~ s2213025$ ./knapsackDP2 10 5
0 0 0 0 0 0
0 250 250 250 250 250
0 250 380 630 630 630
0 250 380 630 630 670
0 250 380 630 770 900
0 250 380 630 770 900
0 250 380 630 770 900
0 250 380 630 770 900
0 250 380 630 770 900
0 250 380 630 770 900
0 250 380 630 770 900
重さ 2 価値 380
重さ 3 価値 520
合計価値 900
```

容量=7

```
azalea02:~ s2213025$ ./knapsackDP2 10 7
0 0 0 0 0 0 0 0
0 250 250 250 250 250 250 250
0 250 380 630 630 630 630 630
0 250 380 630 630 670 800 1050
0 250 380 630 770 900 1150 1150
0 250 380 630 770 900 1150 1150
0 250 380 630 770 900 1150 1150
0 250 380 630 770 900 1150 1150
0 250 380 630 770 900 1150 1150
0 250 380 630 770 900 1150 1150
0 250 380 630 770 900 1150 1150
重さ 1 価値 250
重さ 2 価値 380
重さ 3 価値 520
合計価値 1150
```

容量=10

```
azalea02:~ s2213025$ ./knapsackDP2 10 10
0 0 0 0 0 0 0 0 0 0 0
0 250 250 250 250 250 250 250 250 250 250
0 250 380 630 630 630 630 630 630 630 630
0 250 380 630 630 670 800 1050 1050 1050 1050
0 250 380 630 770 900 1150 1150 1190 1320 1570
0 250 380 630 770 900 1150 1150 1190 1320 1570
0 250 380 630 770 900 1150 1150 1190 1320 1570
0 250 380 630 770 900 1150 1150 1190 1320 1570
0 250 380 630 770 900 1150 1150 1190 1320 1570
0 250 380 630 770 900 1150 1150 1190 1320 1570
0 250 380 630 770 900 1150 1150 1190 1320 1570
重さ 1 価値 250
重さ 2 価値 380
重さ 4 価値 420
重さ 3 価値 520
合計価値 1570
```

より、あっていることがわかる。

考察と感想

今回のアルゴリズムは面白いと思う。ただし、発展課題 2 のサンプルコードに与えられたのは `int** G = makeIntMatrix(n, C);` だが、n を n+1、C を C+1 に変更しないと正しく動作しないことに注意すればよい。