

תכנות דינאמי

Dynamic programming = DP

מה זה תכנות דינאמי?

הרעיון המרכזי מאחורי תכנות דינמי הוא פשוט אך עוצמתי: נשמור את הפתרונות עבור תת-בעיות קטנות, ונשתמש בהם כדי לפתור בעיות גדולות יותר – ובכך נחסוך חישובים מיותרים ונשפר משמעותית את זמן הריצה.

מדרש שם:

השם תכנות דינאמי הומצא על ידי הראשון שפרסם מאמר בנושא – ריצ'רד בלמן, על מנת לגרום לבוס שלו לחשוב שהשיטה מגניבה וחדשנית.

דוגמה ידועה – סדרת פיבונאצ'י

סדרת פיבונאצ'י

דוגמה קלאסית לכך היא סדרת פיבונאצ'י. רבים מכם נחשפו אליה בקורס מבוא למדמ"ח, שם ראיתם כיצד ניתן לחשב את המספר ה-n בעזרת רקורסיה. אך מה הבעיה בגישה הזו?

$$\text{כזכור: } F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

ועל כן הקוד ממבוא למדעי המחשב היה נראה כך:

```
int fib(int n){  
    if(n < 2) return n;  
    return fib(n-1) + fib(n-2);  
}
```

סדרת פיבונאצי רקורסיה

הבעיה היא סיבוכיות הזמן. כל מספר מחושב על ידי קריאה לפונקציה עבור שני המספרים שלפניו, וכל אחת מהקריאות האלו מבצעת בעצמה שתי קריאות נוספות, וכן הלאה. יוצא שסיבוכיות הזמן היא $O(2^n)$ (ולמעשה $O(\varphi^n)$ – לסקרנים קראו יחס הזהב).

$$T(n) = T(n-1) + T(n-2) \geq 2T(n-2) \Rightarrow T(n) = \Omega\left(2^{\frac{n}{2}}\right) \quad \text{החישוב הוא:}$$

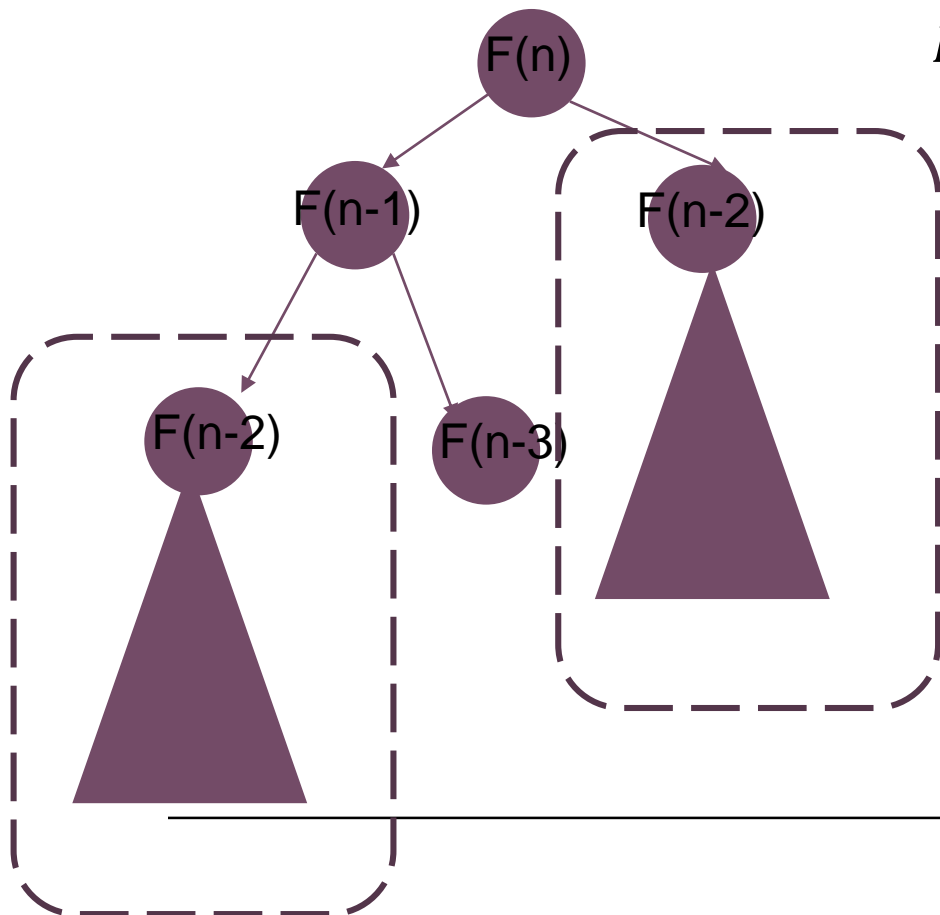
Computing power: *exists*

Recursive Fibonacci:



איך נעשה זאת בצורה יעילה?

ננתח לעומק את הבעיה



- נשים לב כי כשמחשבים את $F(n)$, מחשבים את $F(n-1), F(n-2)$ ואם חישבנו את $F(n-1)$ נכון אז היינו צריכים להשתמש ב $F(n-2)$, ולמה שנחשב אותו פעמיים?

נוכל פשוט לשמור את הערכים במעיין מערך!

לרעיון זה קוראים memoization, וזו לא טעות כתיב.

הקוד החדש

```
#include <bits/stdc++.h>
using namespace std;

const int maxn = 1e5; // shrirooti

vector<int> memoization(n: maxn, value: -1);

int fib(int n){
    if(memoization[n] != -1) return memoization[n];
    if(n < 2) return n;
    return memoization[n] = fib(n: n-1) + fib(n: n-2);
}
```

BU vs TD

- כזכור הרעיון היה לחשב את פיבונאצ'י n ע"י פיבונאצ'י $n-1, n-2$, אך ברקורסיה. דרך נוספת לעשות זאת היא בעזרת לולאה שמחשב את הערכים בסדר עולה ולא בסדר יורד.
 - לשיטה שראינו מקודם קוראים Top-Down, ולשיטה החדשה קוראים Bottom-Up.
 - בכלליות – עדיף להשתמש בתכנות דינאמי בגישה של BU וזאת כי שיטה זו הרבה יותר קלה לדיבוג, אבל ישנם מקרים ששיטת ה-TD פשוטה משמעותית למימוש בסיבוכיות טובה (לצורך הדוגמה, יש פעמים שנוסחאות רקורסיביות הופכות מ- $O(n^2)$ ל- $O(n \log n)$ בעיקר בגלל גישות לזיכרון שאתה צריך בלבד).
-

סדרת פיבונאצ'י BU

נעשה מערך בגודל n . נאתחל את שני המקומות הראשונים ל-1, ומאז נעשה לולאת for שכל פעם מחברת את שני המקומות הקודמים אל המקום הנוכחי.

```
#include <bits/stdc++.h>
#define rep(i, j, k) for(int i = (j); i < (k); i++)
using namespace std;

const int maxn = 1e5; // shirrooti

vector<int> dp(n: maxn, value: -1);

int fib(int n){
    dp[0] = 0, dp[1] = 1;
    rep(i, 2, n+1) dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
}
```

נשים לב ששיטה זו רק משתמשת בשלושה מקומות בכל רגע כלומר ניתן להשתמש ב sliding window :

```
int fib(int n){
    int a, b, c; a = 0, b = 1;
    rep(i, 2, n+1) c = a + b, a = b, b = c;
    return c;
}
```

שאלות לדוגמה

Minimizing Coins

מהי הכמות הקטנה ביותר של מטבעות מבין $a_1, a_2, a_3 \dots, a_n$ איתם ניתן להרכיב את הסכום x ?
חסמי השאלה:

$$1 \leq n \leq 100, \quad 1 \leq x \leq 10^6, \quad 1 \leq a_i \leq 10^6$$

נשים לב שסיבוכיות קבילה תחת חסם הזמן 1_{sec} הינה $O(x \cdot n)$.

דוגמה:

עבור $x = 6$ ומטבעות אפשריים $\{1, 3, 4\}$, הדרך החסכונית ביותר היא להשתמש בשני מטבעות מסוג 3, כלומר: $3 + 3 = 6$.

פתרון

```
using vi = vector<int>;
const int oo = 1e18;

signed main(){
    cin.tie(0)->sync_with_stdio(false), cout.tie(0);
    int n, x, c; cin >> n >> x;
    vi dp(n: x+1, value: oo); dp[0]=0;
    rep(i, 0, n){
        cin >> c;
        rep(j, 0, x-c+1) if(dp[j] != oo) dp[j+c] = min(dp[j+c], dp[j]+1);
    } cout << (dp[x] == oo ? -1 : dp[x]);
}
```

נשים לב שהפתרון האופטימלי עבור x בטוח יהיה הפתרון האופטימלי של x -coin מסוים, ועוד 1 כי נוסיף את המטבע הזה.

לכן נעבור בלולאה על המספרים מ 1 עד n , וכל אחד נעדכן להיות האופטימלי מבין הקודמים אליו בהפרש מטבע אחד.

Coin combinations

כמו מקודם, נניח שניתן להשתמש רק במטבעות בקבוצה $a_1, a_2, a_3, \dots, a_n$, ואנחנו רוצים לשלם ללא עודף על כורסה שעולה x . כמה דרכים שונות יש לעשות את זה, אם יש לו אינסוף מכל מטבע? (הסדר משנה)

For example, if the coins are $\{2, 3, 5\}$ and the desired sum is 9, there are 8 ways:

- $2 + 2 + 5$
- $2 + 5 + 2$
- $5 + 2 + 2$
- $3 + 3 + 3$
- $2 + 2 + 2 + 3$
- $2 + 2 + 3 + 2$
- $2 + 3 + 2 + 2$
- $3 + 2 + 2 + 2$

פתרון

- נעבור על כל סכום, ואם הצלחנו ליצור את הסכום הזה אז נעבור על כל המטבעות, ונוסיף את כמות הדרכים שהייתה ליצור את הסכום הזה לסכום $curr+coin$. (כאשר $curr$ הסכום הנוכחי ו $coin$ המטבע הנוכחי).
- הסיבה היא שעבור כל דרך ליצור את הסכום הקודם אנחנו נוכל להוסיף את המטבע הזה וליצור את הסכום הנוכחי.
- לא נספור אף דרך פעמיים כי לא ניתן ליצור שני סכומים שונים עם אותם מטבעות (לא יכול להיות שתהיה דרך ליצור 4, ודרך ליצור 6, ושתייהן יתנו את אותה דרך ליצור 8).
- נעשה בעזרת מערך, לא לשכוח שניתן ליצור את הסכום 0 בדרך אחת אז נאתחל את 0 ל-1.

```
for(int i = 1; i < x + 1; i++){  
    for(int j = 0; j < n; j++) {  
        if (arr[j] <= i)  
            dp[i] += dp[i - arr[j]];  
    }  
}  
  
cout << dp[x];
```

Paths in a grid

בבעיה הזו, אנחנו מקבלים לוח בגודל $n \times n$ וכל משבצת בו מכילה מספר חיובי v_{ij} . אנחנו מתחילים מהפינה השמאלית-עליונה של הלוח (למעלה משמאל), והמטרה היא להגיע לפינה הימנית-תחתונה (למטה מימין).

מותר לזוז רק **שמאלה** או **למטה** בכל צעד – כלומר, אי אפשר לזוז אחורה או למעלה.

עלינו למצוא מסלול כזה כך שסכום המספרים שנעבור עליהם יהיה **הכי גדול שאפשר**.

לדוגמה, בציור הבא מוצג מסלול אופטימלי שמממש את הדרישה הזו.

סיבוכיות- $O(n^2)$ (גודל הgrid)

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

פתרון

נשתמש במערך דו ממדי dp , כך ש- $dp[i][j]$ ישמור את הסכום המקסימלי שניתן להגיע אליו במשבצת i,j כאשר מותר לזוז רק ימינה או למטה.

המעבר מבוסס על העובדה הפשוטה: כדי להגיע למשבצת מסוימת, ניתן להגיע אליה או מהמשבצת שמעליה או מהמשבצת שמשמאלה. מכאן הנוסחה $dp(i,j) = \max(dp(i,j-1), dp(i-1,j)) + v_{i,j}$. נאתחל את $dp[0][0] = grid[0][0]$ כי זה המקום שממנו מתחילים.

```
for (int y = 1; y <= n; y++) {  
    for (int x = 1; x <= n; x++) {  
        sum[y][x] = max(sum[y][x-1], sum[y-1][x]) + value[y][x];  
    }  
}
```

Not Knapsack

בשאלה זאת נתונים לנו n חפצים עם משקלים שונים, ויש לנו תיק שיכול להכיל W משקל כולל, אנחנו רוצים לבחור חפצים כך שסכומם יהיה קטן מ W , וכמה שקרוב אל W .

סיבוכיות זמן- $O(w \cdot n)$.

דוגמה:

$K = 7, \{6, 2, 3\}$. פה התשובה תהיה 6.

פתרון

- יהיה מערך שיזכור עבור כל סכום אם ניתן ליצור אותו.
- אנחנו לא יכולים לעבור על כל הגדלים האפשריים וכל פעם להוסיף כל חפץ כמו שעשינו ב Minimizing Coins, כי אנחנו לא יודעים אם השתמשנו בחפץ הזה בשביל הסכום הנוכחי. איך נוכל לדעת?
- נעבור על כל חפץ ורק בתוכו נעבור על כל גודל ונעדכן, ככה אנחנו יודעים שלא השתמשנו כבר בחפץ הזה. בעצם נכפה סדר שרירותי והוא ידריך את השימוש שלנו במשאבים.
- בחפץ הא נצטרך ללכת על מערך התשובה שלנו באופן יורד, כדי שלא נשתמש בו פעמיים (התשובות הקודמות שלו יהיו כבר מעלינו, אז לא נחזור אליהן).

```
possible[0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = W; x >= 0; x--) {
        if (possible[x]) possible[x+w[k]] = true;
    }
}
```

Knapsack

- אתם שודדים חנות שיש בה חפצים. לכל חפץ $i \in \{1, \dots, n\}$ יהיה משקל w_i ומחיר c_i ונרצה בהינתן תיק גנבים שמסוגל להכיל עד W משקל כולל את הסכום המחירים המקסימלי שניתן להשיג ע"י כמה חפצים שסכום משקלם לא עולה על W . קבוצת החפצים מסומנת H עבור hafatzim.
- בניסוח מתמטי:

$$\max_{\substack{S \subseteq H \\ \sum_{i \in S} w_i \leq W}} \sum_{i \in S} c_i$$

פתרון

• נסמן:

המחיר המקסימלי שניתן להשיג עם סכום משקלים x ובעזרת תת קבוצה של $\{1, \dots, i\}$ $dp(x, i)$

הפתרון הוא כמובן (בהינתן המערך המחושב הזה): $\max_{0 \leq x \leq W} dp(x, n)$

כעת נותר לחשוב כיצד נחשב את המערך המופלא הזה?

דבר נוח יהיה לכפות סדר שרירותי על החפצים כדי שיהיה ברור מה זה "האיבר ה- i "

נשים לב שבפתרון האופטימלי עבור $dp(x, i)$ עבור x, i כלשהם, אנחנו יכולנו או להוסיף לתיק הגנבים את החפץ ה- i או להתעלם ממנו.

אם התעלמנו, אז התשובה תהיה לזו של תת הקבוצה $\{1, \dots, i-1\}$ עם המשקל x , כלומר $dp(x, i-1)$.

אם הוספנו, אז התשובה תהיה המחיר שחפץ i הוסיף לנו (c_i) ועוד המחיר האופטימלי של תת הקבוצה $\{1, \dots, i-1\}$ עם המשקל $x - w_i$.

כתוצאה מכך הגענו לנוסחה $dp(x, i) = \max\{dp(x, i-1), dp(x - w_i, i-1) + c_i\}$, ולאחר בדיקת גבולות וקצת דיבוג שאלה זו נפתרה.
