Data Structures II: Dynamic Programming

Mauricio Toro
Department of Systems and Informatics
Universidad Eafit

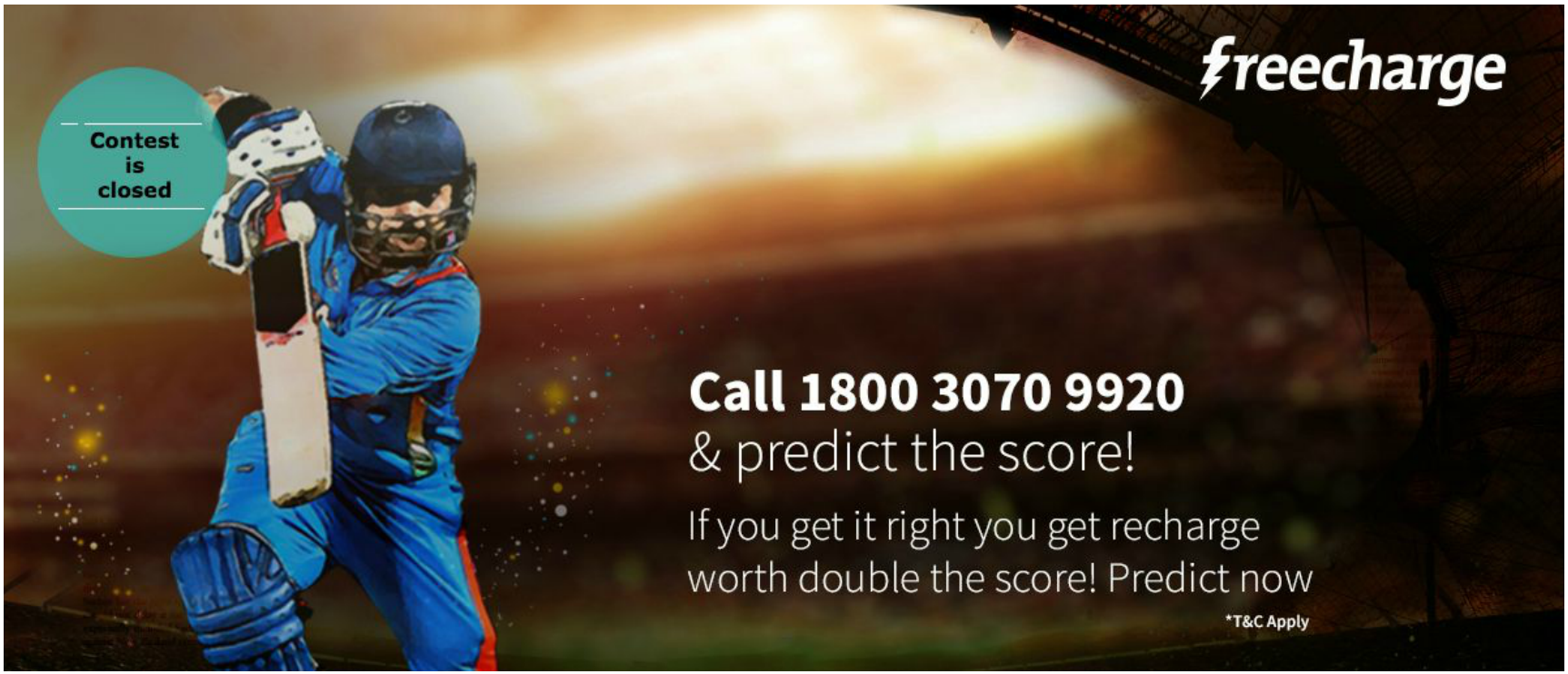
Cocktail of the day: Gin Tonic

Disclaimer: Keep alcohol out of the hands of minors.

Cocktail of the day: Gin Tonic

- 50 ml of tonic water
- 30 ml of gin

Score Predictors use dynamic programming



https://en.wikipedia.org/wiki/WASP_%28cricket_calculation_tool%29

Dynamic Programming

- Dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems.
- It is applicable to problems exhibiting the properties of
  - overlapping subproblems and
  - optimal substructure
- The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations

Taken from Wikipedia

Dynamic Programming

- Dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems.
- It is applicable to problems exhibiting the properties of
  - overlapping subproblems and
  - optimal substructure
- The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations

Taken from Wikipedia

Dynamic Programming

- Dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems.
- It is applicable to problems exhibiting the properties of
  - overlapping subproblems and
  - optimal substructure
- The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations

Taken from Wikipedia

Factorial equation

Compute the factorial of n

$$n! = \begin{cases} 1 & if \quad n = 1 \\ n * (n-1)! & if \quad n > 1 \end{cases}$$

Recursive Factorial

Print the factorial of numbers from 1 to 10

```
private int factorialAUX(int n){
   if (n==1)
    return n;
   else
    return n*factorialAUX(n-1);
}

public void factorial(){
   for (int i = 0; i < 10; i++)
     System.out.println(factorialAUX(i+1));
}
```

Recursive Factorial Complexity

Recursive Factorial Complexity

The complexity of $O(n^2)$

Dynamic Programming Factorial

Print the factorial of numbers from 1 to 10

```
public void factorial() {
   int[] factorials = new int[10];
   for (int i = 0; i < 10; i++) {
      if (i==0)
       factorials[i] = 1;
      else {
        factorials[i] = factorials[i-1]*i;
        System.out.println(factorials[i]);
      }
   }
}
```

Dynamic Programming Factorial Complexity

The complexity of $O(n)$

Fibonacci equation

Print the nth Fibonacci term

$$fibo(n) = \begin{cases} n & if \quad n \leq 1 \\ fibo(n-1) + fibo(n-2) & if \quad n > 1 \end{cases}$$

Applications of Fibonacci

https://plus.maths.org/content/life-and-numbers-fibonacci

Recursive Fibonacci

Print the nth Fibonacci term

```
public int fibo(int n){
   if (n<=1)
    return n;
   else
    return fibo(n-1)+fibo(n-2);
}
```

Recursive Fibonacci Complexity

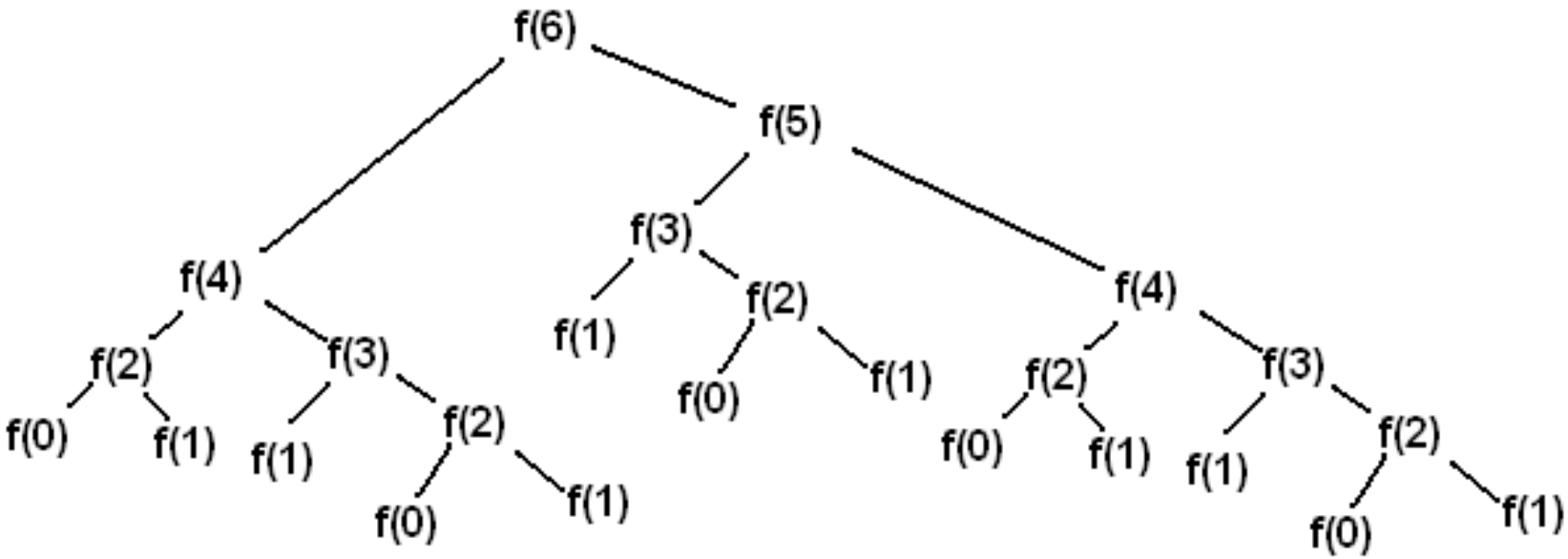


Figure: Recursive Fibonacci execution tree, taken from Wikibooks

Recursive Fibonacci Complexity

The complexity is $O(2^n)$

Dynamic Programming Fibonacci

Print the nth Fibonacci term

```
public int fibo(int n) {
   int[] fibos = new int[n+1];
   for (int i = 0; i <= n; i++) {
      if (i<=1)
       fibos[i] = i;
      else
        fibos[i] = fibos[i-1]+fibos[i-2];
   }
   return fibos[n];
}
```

Dynamic Programming Fibonacci Complexity

The complexity is $O(n)$

Levenshtein distance

- Levenshtein distance is a string metric for measuring the difference between two sequences.
- The Levenshtein distance between two words is the minimum number of single-character edits required to change one word into the other.
  - Insertions
  - Deletions
  - Substitutions

https://www.youtube.com/watch?v=dUSqwTC8TM8
Taken from Wikipedia

Levenshtein distance

- Levenshtein distance is a string metric for measuring the difference between two sequences.
- The Levenshtein distance between two words is the minimum number of single-character edits required to change one word into the other.
  - Insertions
  - Deletions
  - Substitutions

`https://www.youtube.com/watch?v=dUSqwTC8TM8`
Taken from Wikipedia

Examples of Levenshtein Distance

- The Levenshtein distance between "kitten" and "sitting" is 3
  1. kitten $\rightarrow$ sitten (substitution of "s" for "k")
  2. sitten $\rightarrow$ sittin (substitution of "i" for "e")
  3. sittin $\rightarrow$ sitting (insertion of "g" at the end).

Taken from Wikipedia

Applications of Levenhtein Distance

- Spell checkers
- Correction systems for optical character recognition
- Software to assist natural language translation

Note: The strings could come from a dictionary
Taken from Wikipedia

Definition of the Levenshtein distance

Mathematically, the Levenshtein distance between two strings a and b is given by $lev_{a,b}(|a|, |b|)$

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Figure: Levenshtein distance, taken from Wikipedia

Definition of the Levenshtein distance

Mathematically, the Levenshtein distance between two strings a and b is given by $lev_{a,b}(|a|, |b|)$

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Figure: Levenshtein distance, taken from Wikipedia

Definition of the Levenshtein distance

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Figure: Levenshtein distance, taken from Wikipedia

1. The first element in the minimum corresponds to deletion (from a to b), the second to insertion, and the third to match or mismatch, depending on whether the respective symbols are the same.
2. $1_{(a_i \neq b_j)}$ is equal to 0 when $a_i = b_j$, and equal to 1 otherwise.

Recursive Levenshtein Distance

```
int LevenshteinDistance(string s,int len_s,string t,int len_t){
  if (len_s == 0) return len_t;
  if (len_t == 0) return len_s;
  if (s[len_s-1] == t[len_t-1])
      cost = 0;
  else
      cost = 1;
  return minimum(LevenshteinDistance(s,len_s-1,t,len_t)+1,
          LevenshteinDistance(s,len_s,t,len_t-1)+1,
          LevenshteinDistance(s,len_s-1,t,len_t-1)+cost);
}
```

Taken from Wikipedia

Recursive Levenshtein Distance Complexity

Recursive Levenshtein Distance Complexity

The complexity is $O(3^n)$, where n is the length of the longest string

What if we make a table? (2)

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

What if we make a table? (3)

Levenshtein demo:

`http://www.let.rug.nl/kleiweg/lev/`

Dynamic Programming Levenshtein Distance

Dynamic Programming Levenshtein Distance

```
int LevenshteinDistance(char s[1..m], char t[1..n]) {
  declare int d[0..m, 0..n]
  set each element in d to zero
  for i from 1 to m do d[i, 0] := i
  for j from 1 to n do d[0, j] := j
  for j from 1 to n
      for i from 1 to m
          if s[i] = t[j] then
            d[i, j] := d[i-1, j-1] //no operation required
          else
            d[i, j] := minimum(d[i-1, j] + 1,// a deletion
                               d[i, j-1] + 1,     //an insertion
                               d[i-1, j-1] + 1)// a substitution
  return d[m, n]
} //Taken from Wikipedia
```

What if we make a table?

- The Levenshtein distance between "kitten" and "sitting" is 3
  1. kitten $\rightarrow$ sitten (substitution of "s" for "k")
  2. sitten $\rightarrow$ sittin (substitution of "i" for "e")
  3. sittin $\rightarrow$ sitting (insertion of "g" at the end).

Taken from Wikipedia

Dynamic Programming Levenshtein Distance

The complexity is $O(nm)$, where n is the length of the longest string and m is the length of the other string

References

- Please learn how to reference images, trademarks, videos and fragments of code.
- Avoid plagiarism

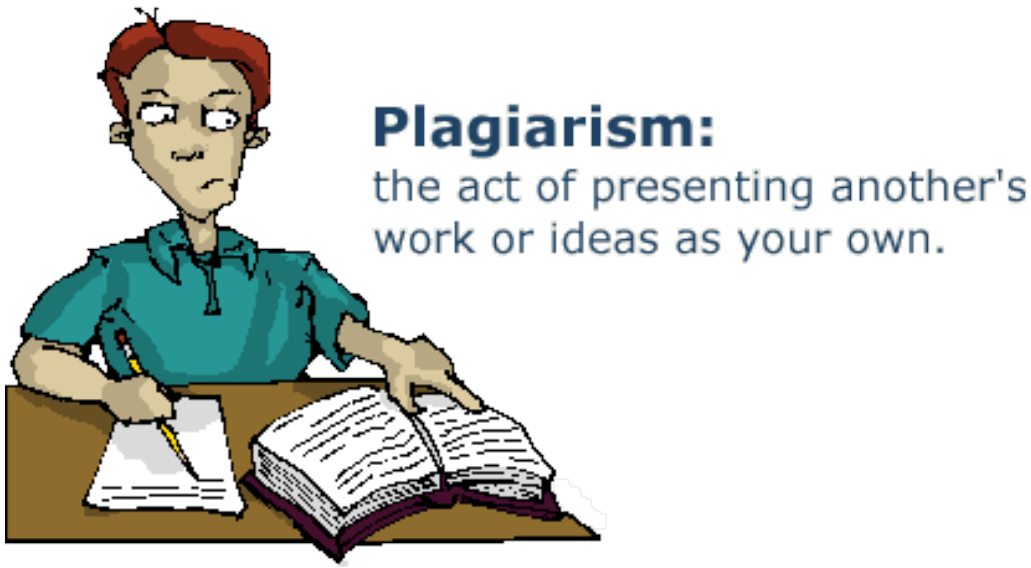


Figure: Figure about plagiarism, University of Malta [Uni09]

References

University of Malta.

Plagarism — The act of presenting another's work or ideas as your own, 2009.

[Online; accessed 29-November-2013].

Further reading

- R.C.T Lee, Introduction to the analysis and design of algorithms, Chapter 7.