dp

May 18, 2020

1 Dynamic Programming Examples

1.1 Fibonacci

1.1.1 Recursive

[1]:
```
def fibo(n):
    if n < 2:
        return n
    return fibo(n-1) + fibo(n-2)
```

[2]:
```
import time
def measure(f):
    start = time.time()
    f()
    end = time.time()
    return end-start
```

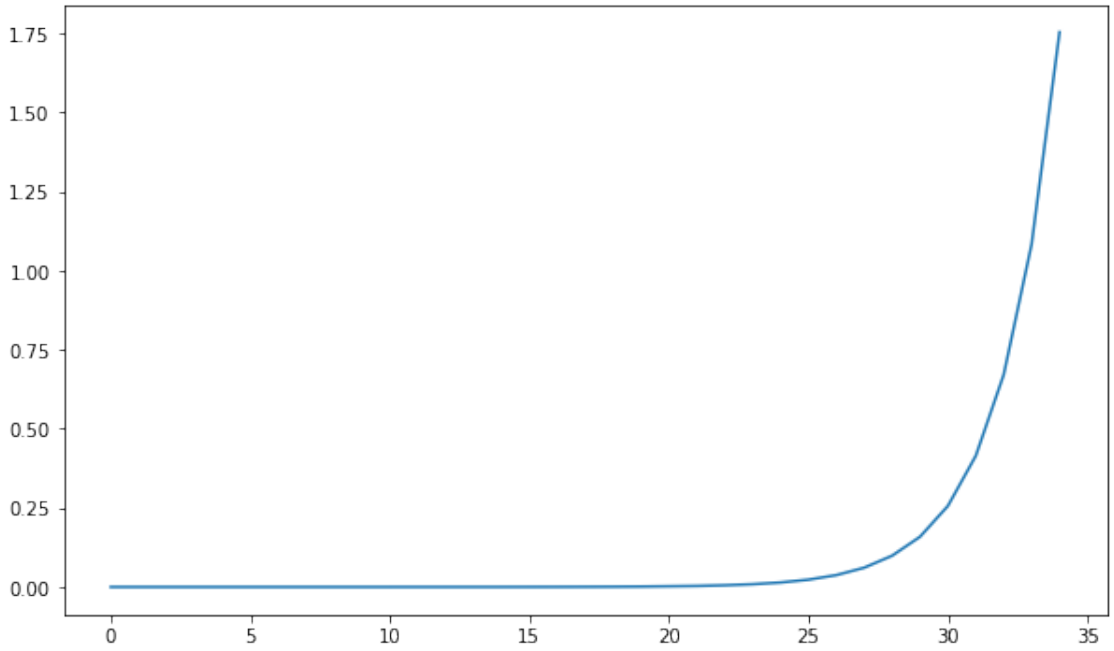
[3]:
```
measure(lambda: print(fibo(35)))
```

```
9227465
```

[3]:
```
2.817490816116333
```

[4]:
```
import matplotlib.pyplot as plt
def show_time(f,n):
    duration = [measure(lambda: f(i)) for i in range(0,n)]
    plt.figure(figsize=(10,6))
    plt.plot(duration)
    plt.show()
```

[5]:
```
show_time(fibo,35)
```



1.1.2 Memoization

[6]:
```
def fibo_m(n,d={}):
    if n in d:
        return d[n]
    else:
        if n < 2:
            f = n
        else:
            f = fibo_m(n-1,d) + fibo_m(n-2,d)
        d[n] = f
        return f
```

[7]:
```
print(fibo_m(35))
```

```
9227465
```

[8]:
```
measure(lambda: fibo_m(35))
```

[8]: 2.86102294921875e-06

1.1.3 Table-based approach

[9]:
```
def fibo_t(n):
    m=[0,1]
    for i in range (2,n+1):
        m.append(m[i-1]+m[i-2])
    return m[n]
```

[10]:
```
print(fibo_t(5000))
```

```
38789684543883256337019163083259053120821277146462451061605972148955501390440370
97010822916462210669479293452858882973813483102008954982940361430156911478938364
21656394410691021450563413370655865623825465670071252592990385493381392883637834
75189087629707120333370529231076930085180938498018038478139967488817655546537882
91644268912980384613778969021502293082475666346224923071883324803280375039130352
90330450584270114763524227021093463769910400671417488329842289149127310405432875
32980442736768229772449877498745556919077038806370468327948113589737399931101062
19308149018570815397854379195305617510761053075688783766033667355445258844886241
61921055345749367589784902798823435102359984466393485325641195222185956306047536
46454707603309024208063825849291564528762915757591423438091423029174910889841552
09854432486594079793571316841692868039545309545388698114665082066862897420639323
43848846524098874239587380197699382031717420893226546887936400263079778005875912
96713896342142525791168727556003603113705477547246046399875880469851784086743828
63125
```

1.2 Rod Cutting

[11]:
```
import random
def make_cost(n):
    random.seed(n)
    cost = [0] * (n+1)
    for i in range(1,n+1):
        cost[i] = cost[i-1]+random.uniform(0,1)
    return cost
```

[12]:
```
print(make_cost(10))
```

```
[0, 0.5714025946899135, 1.0002916493650282, 1.5783829504994986,
1.7844811826390004, 2.5978024339963204, 3.421391306529766, 4.074863840430941,
4.235093396949761, 4.755762756589686, 5.083535568211779]
```

1.2.1 Recursive

[13]:
```
def optimal_cut_r(cost,n):
    if n == 0:
        return 0
    cuts = [cost[n-i] + optimal_cut_r(cost,i) for i in range(0,n)]
    return max(cuts)
```

```
def optimal_cut_R(cost):
    return optimal_cut_r(cost,len(cost)-1)
```

[14]:
```
print(make_cost(10))
print(optimal_cut_R(make_cost(10)))
```

```
[0, 0.5714025946899135, 1.0002916493650282, 1.5783829504994986,
1.7844811826390004, 2.5978024339963204, 3.421391306529766, 4.074863840430941,
4.235093396949761, 4.755762756589686, 5.083535568211779]
5.789071624500682
```

[15]:
```
measure(lambda: print(optimal_cut_R(make_cost(23))))
```

```
21.533046577436288
```

[15]: 3.810856819152832

1.2.2 Memoization

[16]:
```
def optimal_cut_m(cost,n,m):
    if n == 0:
        return 0
    if n in m:
        return m[n]
    cuts = [cost[n-i] + optimal_cut_m(cost,i,m) for i in range(0,n)]
    m[n] = max(cuts)
    return m[n]

def optimal_cut_M(cost):
    return optimal_cut_m(cost,len(cost)-1,{})
```

[17]:
```
print(optimal_cut_R(make_cost(10)))
print(optimal_cut_M(make_cost(10)))
```

```
5.789071624500682
5.789071624500682
```

[18]:
```
measure(lambda: print(optimal_cut_M(make_cost(23))))
```

```
21.533046577436288
```

[18]: 0.0004048347473144531

1.2.3 Table

[19]:
```
def optimal_cut_t(cost,n):
    m = [0] * (n+1)
    for i in range(1,n+1):
        cuts = [cost[i-j] + m[j] for j in range(0,i)]
        m[i] = max(cuts)
    return m[n]

def optimal_cut_T(cost):
    return optimal_cut_t(cost,len(cost)-1)
```

[20]:
```
print(optimal_cut_T(make_cost(10)))
```

```
5.789071624500682
```

1.2.4 Computing the Cut

[21]:
```
def optimal_cut_l(cost):
    n = len(cost)
    m = [0] * n
    l = [0] * n
    for i in range(1,n):
        m[i] = cost[i] # omitted + m[0]
        for j in range(1,i):
            c = cost[i-j]+m[j]
            if c > m[i]:
                m[i] = c
                l[i] = j
    return (m[n-1],l)

def print_cut(cut,cost):
    print("best cost:",cut[0])
    l = cut[1]
    n = len(l)-1
    while n > 0:
        print(n-l[n], "(",cost[n-l[n]],")")
        n = l[n]
    print()
```

[22]:
```
cost = make_cost(10)
print(cost)
cut = optimal_cut_l(make_cost(10))
print(cut[1])
print_cut(cut,cost)
```

```
[0, 0.5714025946899135, 1.0002916493650282, 1.5783829504994986,
1.7844811826390004, 2.5978024339963204, 3.421391306529766, 4.074863840430941,
```

```
4.235093396949761, 4.755762756589686, 5.083535568211779]
[0, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3]
best cost: 5.789071624500682
7 ( 4.074863840430941 )
1 ( 0.5714025946899135 )
1 ( 0.5714025946899135 )
1 ( 0.5714025946899135 )
```

1.3 Rabbits

[23]:
```
import random
class Grid:
    def __init__(self,n,m):
        """n rows (y), m columns (x)"""
        random.seed(7)
        self.south = [[random.randrange(10) for i in range(m)] for j in␣
↪range(n-1)]
        self.east = [[random.randrange(10) for i in range(m-1)] for j in␣
↪range(n)]
```

1.3.1 Recursive

[24]:
```
def best_path_r(grid,y,x):
    east = best_path_r(grid,y,x+1)+grid.east[y][x] if (x<len(grid.east[y]))␣
↪else 0
    south = best_path_r(grid,y+1,x)+grid.south[y][x] if (y<len(grid.south))␣
↪else 0
    return max(east,south)

def best_path(n,m):
    return best_path_r(Grid(n,m),0,0)

measure(lambda: print(best_path(14,14)))
```

```
184
```

[24]:
```
18.234222650527954
```

1.3.2 Memoization

[25]:
```
def best_path_m(grid,y,x,m):
    if (x,y) in m:
        return m[(x,y)]
    east = best_path_m(grid,y,x+1,m)+grid.east[y][x] if (x<len(grid.east[y]))␣
↪else 0
```

```
    south = best_path_m(grid,y+1,x,m)+grid.south[y][x] if (y<len(grid.south))␣
 ↪else 0
    m[(x,y)] = max(east,south)
    return m[(x,y)]

def best_path_M(n,m):
    return best_path_m(Grid(n,m),0,0,{})


measure(lambda: print(best_path_M(14,14)))
```

```
184
```

[25]:
```
0.0009031295776367188
```

1.3.3 Table

[26]:
```
def best_path_t(grid,n,m):
    M = [[0]*m for j in range(n)]
    # do not use [[0]*m]*n because it returns a matrix of references to same row
    for y in reversed(range(0,n)):
        for x in reversed(range(0,m)):
            east = M[y][x+1]+grid.east[y][x] if (x<len(grid.east[y])) else 0
            south = M[y+1][x]+grid.south[y][x] if (y<len(grid.south)) else 0
            M[y][x] = max(east,south)
    return M[0][0]

def best_path_T(n,m):
    return best_path_t(Grid(n,m),n,m)


measure(lambda: print(best_path_T(14,14)))
```

```
184
```

[26]:
```
0.00060272216796875
```