X1-ComputationalComplexity-notebook

May 20, 2021

1 Nested loops

The following two functions compute sum and product of matrices, respectively.

By counting the nested loops it is easy to see that `add()` is $O(n^2)$ while `prod()` is $O(n^3)$.

[37]:
```
from random import randint
import time

def add(A, B):
    S = [[0] * len(A) for i in range(len(A))]
    for i in range(len(A)):
        for j in range(len(A)):
            S[i][j] = A[i][j] + B[i][j]
    return S

def prod(A, B):
    S = [[0] * len(A) for i in range(len(A))]
    for i in range(len(A)):
        for j in range(len(A)):
            for k in range(len(A)):
                S[i][j] = S[i][j] + A[i][k] * B[k][j]
    return S

N = 10
A = [ [randint(0,100) for i in range(N)] for j in range(N) ]
B = [ [randint(0,100) for i in range(N)] for j in range(N) ]

t0 = time.process_time()
add(A,B)
t1 = time.process_time()
prod(A,B)
t2 = time.process_time()

print("Time for add: ", t1-t0)
print("Time for prod:", t2-t1)
```

```
Time for add:  0.00012074300000008975
Time for prod: 0.00036587199999971176
```

2 Sorting a list, slow version

The following code implements a slow version of the so-called *insertion sort* alogithm

Complexity: $O(n^2)$.

[1]:
```
from random import randint
import time

def correct_position(e, S):
    for i in range(len(S)):
        if S[i] > e:
            return i
    return len(S)

def sort_list(L):
    S = []
    for e in L:
        cp = correct_position(e, S)
        S.insert(cp, e)
    return S

N = 10000
L = [randint(0,10**9) for i in range(N)]

t0 = time.process_time()
sort_list(L)
t1 = time.process_time()

print("Running time:", t1-t0)
```

```
Running time: 1.1191449070000001
```

3 Binary search

The following code implements a binary search.

Complexity: $O(\log_2(n))$

[3]:
```
from random import randint
import time

def binary_search(e, S, start, end):
    if start == end:
        return start
    midpoint = (start+end) // 2
    if e < S[midpoint]:
        return binary_search(e, S, start, midpoint)
    else:
```

```
        return binary_search(e, S, midpoint+1, end)

N = 1000000
L = [randint(0,10**9) for i in range(N)]
e = randint(0,10**9)

t0 = time.process_time()
L.sort() # Using Python's sort()
t1 = time.process_time()
i = binary_search(e, L, 0, len(L))
t2 = time.process_time()
print("The correct position of e =", e, "in L is:")
print("...", L[i-2], L[i-1], "e", L[i], L[i+1], "...")
print("")
print("Time for sorting:  ", t1-t0)
print("Time for searching:", t2-t1)
```

```
The correct position of e = 216197744 in L is:
… 216196218 216197540 e 216198673 216198962 …

Time for sorting:   0.26413054400000036
Time for searching: 9.616099999965044e-05
```

4 Sorting a list, fast version (with binary_search)

The following code uses the function `binary_search()` above instead of `correct_position()` in our insertion sort algorithm.

Complexity: $O(n \log_2(n))$

[69]:
```
from random import randint
import time

def binary_search(e, S, start, end):
    if start == end:
        return start
    midpoint = (start+end) // 2
    if e < S[midpoint]:
        return binary_search(e, S, start, midpoint)
    else:
        return binary_search(e, S, midpoint+1, end)

def sort_list(L):
    S = []
    for e in L:
        cp = binary_search(e, S, 0, len(S)) # Changed here
        S.insert(cp, e)
```

```
    return S

N = 10000
L = [randint(0,10**9) for i in range(N)]

t0 = time.process_time()
sort_list(L)
t1 = time.process_time()

print("Running time:", t1-t0)
```

```
Running time: 0.03710268399998995
```

5 Fast exponentiation

The following cell contains two functions for computing a^n (n non-negative integer): a slow one that runs in $O(n)$ and a fast one that runs in $O(\log_2(n))$. We compare these two also with Python's built-in operator `**`.

Complexity: $O(n)$ for the slow algorithm, $O(\log_2(n))$ for the other two.

[30]:
```
import time

def slow_power(a, n):
    r = 1
    for i in range(n):
        r = r * a
    return r

def fast_power(a, n):
    if n == 0:
        return 1
    if n%2 == 0:
        return fast_power(a*a, n//2)
    else:
        return a * fast_power(a, n-1)

a = 1.00000001
n = 100000000

t0 = time.process_time()
print(slow_power(a, n))
t1 = time.process_time()
print(fast_power(a, n))
t2 = time.process_time()
print(a**n)
t3 = time.process_time()
```

```python
print("Time for slow_power():", t1-t0)
print("Time for fast_power():", t2-t1)
print("Time for Python's **: ", t3-t2)
```

```
2.71828179834636
2.7182817863957984
2.7182817983473577
Time for slow_power(): 3.234879998000004
Time for fast_power(): 9.059099999575437e-05
Time for Python's **:  0.00010159500000384014
```

6 Fast gcd

Complexity: $O(\log_2(n))$

[31]:
```python
import time

def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a%b)

t0 = time.process_time()
print(gcd(155275387236018, 572335397352432))
t1 = time.process_time()

print("Running time:", t1-t0)
```

```
126
Running time: 0.00017707599999994272
```

7 Fibonacci numbers

In the following cell there are two functions that compute the n-th Fibonacci number. They are almost the same, but the second one memorizes the results in a list to avoid computing them multiple times, and it is much much faster.

Complexity: $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) \sim O(1.6^n)$ for the slow version, $O(n)$ for the fast version.

[37]:
```python
import time

F_memorized = [-1] * (10**6)

def F_slow(n):
    if n <= 1:
```

```python
            return n
        else:
            return F_slow(n-1) + F_slow(n-2)

def F_fast(n):
    if F_memorized[n] == -1:
        if n <= 1:
            F_memorized[n] = n
        else:
            F_memorized[n] = F_fast(n-1) + F_fast(n-2)

    return F_memorized[n]

n = 35

t0 = time.process_time()
print(F_slow(n))
t1 = time.process_time()
print(F_fast(n))
t2 = time.process_time()

print("Time for F_slow:", t1-t0)
print("Time for F_fast:", t2-t1)
```

```
9227465
9227465
Time for F_slow: 2.3301570169999906
Time for F_fast: 8.848800000293977e-05
```

[ ]:

```python

```