Computational Complexity - Pseudocode and Recursions

Nicolò Felicioni[1]

Dipartimento di Elettronica e Informazione
Politecnico di Milano

nicolo . felicioni @ polimi . it

May 16, 2021

[1] Mostly based on Nicholas Mainardi's material.

More complexity analysis?

Asymptotic complexity assessment

- In the following slides we will assess the asymptotic complexity of some algorithms using the constant cost criterion
- This is justified by the fact that all the involved variables are considered to be bounded in size to, at most, a machine word, thus all the atomic operations are effectively $O(1)$
- This lesson also tackles the issue of analysing the complexity of algorithms involving recursive function calls

Pseudocode Time Complexity - Example 1

```
P(n)
1  s ← 0
2  for i ← 1 to n
3        do j ← 1
4           while j < n
5                 do k ← 1
6                    while k < n
7                          do s++
8                             k ← 3 * k
9                    j ← 2 * j
```

- The first cycle runs n times
- The first **while** loop runs until $j < n$, and j is doubled each time, that is until $2^h < n \Rightarrow h < \log_2(n)$
- The second **while** loop runs until $k < n$, and k is triplied each time, that is until $3^h < n \Rightarrow h < \log_3(n)$
- Total complexity: $n * \log_2(n) * \log_3(n) = \Theta(n \log^2(n))$

Pseudocode Time Complexity - Example 2

```
P(n)
1   i ← 2
2   j ← 0
3   while i < n
4         do j++
5            i ← i * i
```

- How many times is the loop executed?
- i starts from 2 and it is squared at each iteration
- Th sequence of i values is: $2, 4, 16, 256, 2^{16} \ldots$
- The loop body is executed until $i < n$, that is until $2^{2^h} < n \Rightarrow 2^h < \log_2(n) \Rightarrow h < \log_2(\log_2(n))$
- Total time complexity: $\Theta(\log(\log(n)))$

Pseudocode Time Complexity - Example 3

```
P(n)
 1   a ← 0
 2   j ← 1
 3   k ← 0
 4   while j < n
 5         do k++
 6            for i ← 1 to k
 7                  do h ← 2
 8                     while h < 2^n
 9                           do a++
10                              h ← h * h
11            j ← 2 * j
```

- The number of iterations of the **for** loop is dependent on the outer loop
  - ↪ it depends on the number of times k is incremented

Pseudocode Time Complexity - Example 3

Complexity Estimation in case of Dependent Nested Cycles

- the outer loop is executed $\log(n)$ times, thus k can be incremented up to $\log(n)$
- At each iteration, the **for** loop runs k times. Total number of executions of **for** loop body?
- $\sum_{k=1}^{\log(n)} k = \frac{\log(n)*(\log(n)+1)}{2} = \Theta(\log^2(n))$
- Each of these executions has another loop
- This loop is independent from the previous two. How many times does it run?
- the loop runs until $h < 2^n$, with h being squared at each iterations, thus it runs until $2^{2^m} < 2^n \Rightarrow 2^m < n \Rightarrow m < \log(n)$
- Total time complexity is $\Theta(\log^3(n))$

Pseudocode Time Complexity - Example 4

```
P(n)
1   sum ← 0
2   for i ← 1 to log(n)
3         do j ← 2
4            while j < 2^n
5                  do sum++
6                     j ← 2 * j
```

- The outer loop is executed $\log(n)$ times
- The inner loop is executed until $j < 2^n$, with j being doubled at each iteration, thus it runs until $2^h < 2^n \Rightarrow h < n$
- Total time complexity: $\Theta(n \log(n))$

Pseudocode Time Complexity - Example 4

Consider this variant:

$\mathrm{P}(n)$

```
1   sum ← 0
2   for i ← 1 to log(n)
3         do j ← 2
4            while j < 2^n
5                  do sum ← sum + F(n)
6                     j ← 2 * j
```

where $T(\mathrm{F}(n)) \in \Theta(\log(n))$

- The number of iterations of the loops are the same as the original version $\Rightarrow \Theta(n \log(n))$
- However, each inner loop body has a logarithmic cost
- Thus, total time complexity is $\Theta(n \log(n) \log(n)) = \Theta(n \log^2(n))$

Pseudocode Time Complexity - Example 4

Another variant:

$\mathrm{P}(n)$

```
1   sum ← 0
2   for i ← 1 to log(n)
3         do j ← 2
4            while j < 2^n
5                  do sum ← sum + F(i)
6                     j ← 2 * j
```

where $T(\mathrm{F}(n)) \in \Theta(\log(n))$

- This time the complexity of the inner loop body is dependent from the outer cycle
- The total complexity is $T(n) = \sum_{i=1}^{\log(n)} n \log(i)$

Pseudocode Time Complexity - Example 4

A bit of math

- $T(n) = \sum_{i=1}^{\log(n)} n \log(i) = n \sum_{i=1}^{\log(n)} \log(i) = n \log(\prod_{i=1}^{\log(n)} i)$, by applying logarithm properties
- $T(n) = n \log(\prod_{i=1}^{\log(n)} i) = n \log(\log(n)!)$
- Recall the Stirling approximation to get rid of the factorial: $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$
- $T(n) = n \log(\log(n)!) = n \log(\sqrt{2\pi \log(n)}(\frac{\log(n)}{e})^{\log(n)})$
- Apply logarithm properties:
  $T(n) = n \log(\sqrt{2\pi \log(n)}(\frac{\log(n)}{e})^{\log(n)}) =$
  $\Theta(n \log(\sqrt{2\pi \log(n)}) + n \log(\log(n)^{\log(n)})) =$
  $\Theta(n \log(\log(n)^{\frac{1}{2}}) + n \log(n) \log(\log(n))) =$
  $\Theta(n \frac{1}{2} \log(\log(n)) + n \log(n) \log(\log(n))) =$
  $\Theta(n \log(n) \log(\log(n)))$

Computing exponentiations: m^n

Straightforward implementation

- We want to compute m^n: the following function does so:

$\textsc{Exp}(n, m)$

```
1  res ← 1
2  for i ← 1 to n
3        do res ← res × m
4  return res
```

- Complexity? the loop runs exactly n times, thus $O(n)$
- Is there a fastest algorithm to compute exponentiation?

Way faster exponentiations

Another look at the exponent

- We can rewrite the exponent n as a $t = \lceil \log_2(n) \rceil$ bit binary number $n = (b_{t-1} \dots b_0)$
- Recalling common mathematical properties
  $m^n = m^{(2^{t-1}b_{t-1}+2^{t-2}b_{t-2}+\dots+2^0 b_0)} =$
  $m^{2^{t-1}b_{t-1}} \cdot m^{2^{t-2}b_{t-2}} \cdot \dots \cdot m^{2^1 b_1} \cdot m^{2^0 b_0}$
- We note that m^n is actually the product of all the m^{2^i} for which the i-th bit is one
- All the m^{2^i} values can be obtained by squaring $m^{2^{i-1}}$, at the cost of a single multiplication
- Exploiting this observation we can design an algorithm which computes them and multiplies together only the ones where $b_i = 1$

Right-to-Left Square and Multiply

Algorithm and Complexity

$\text{SMExp}(n, m)$

1 $res \leftarrow 1$
2 $tmp \leftarrow m$
3 **for** $i \leftarrow 0$ **to** $(t-1)$
4 **do if** $n \,\&\, (1 << i) \neq 0$ ▷ It is the same as $b_i = 1$
5 **then** $res \leftarrow res \times tmp$
6 $tmp \leftarrow tmp \times tmp$
7 **return** res

- Complexity? The loop body is run $t = \lceil \log_2(n) \rceil$ times: $O(log(n))$, way faster than before!

Computing Hamming Weight

Straightforward Method

- The Hamming Weight of n is defined as the number of ones in its binary representation

HW(n)

1 $tmp \leftarrow n$
2 $hw \leftarrow 0$
3 **while** $tmp > 0$
4 **do if** $tmp \bmod 2 = 1$
5 **then** $hw \leftarrow hw + 1$
6 $tmp \leftarrow \lfloor \frac{tmp}{2} \rfloor$
7 **return** hw

- Complexity? Loop until the result of the repeated integer division by 2 of the input is 0
- The loop is run $log_2(n)$ times, thus $O(\log(n))$

Computing HW, Kernighan Style

Arithmetic helps

- Is the previous method the best we can do? No!

HWKernighan(n)

```
1  tmp ← n
2  hw ← 0
3  while tmp > 0
4        do hw ← hw + 1
5           tmp ← tmp&(tmp − 1)
6  return hw
```

- Line 5 effectively removes only the least significant digit set to 1 of the number in $O(k)$ exploiting the effect of the borrows
- Thus the loop runs exactly as many times as the Hamming weight of the input, thus $\Theta(HW(n))$

Is this a perfect power?

Given n, is it a perfect power $n = a^x$, $x \in \mathbb{N} \setminus \{0, 1\}$?

Strategy 1:

- We try all possible bases k, from $k = 2$ to $\sqrt{n}$
- How many exponents exp do we need to try? until $k^{exp} \leq n \Rightarrow exp \leq \log_k(n)$
- Complexity: $\sum_{k=2}^{\sqrt{n}} \log_k(n) = O(\sqrt{n} \log(n))$

Strategy 2:

- We try all possible exponents exp, up to $\log_2(n)$
- How many bases do we need to try for each exp? until $k^{exp} \leq n \Rightarrow k \leq \sqrt[exp]{n}$
- At each iteration, we perform an exponentiation $O(\log(exp))$
- Complexity: $\sum_{exp=2}^{\log_2(n)} \sqrt[exp]{n} \log(exp) = \sqrt{n} + \sum_{exp=3}^{\log_2(n)} \sqrt[exp]{n} \log(exp)$. The second term is $O(\sqrt[3]{n} \log(n) \log(\log(n)))$ and thus the complexity is $\sqrt{n} + O(\sqrt[3]{n} \log(n) \log(\log(n))) = O(\sqrt{n})$

Is this a perfect power?

A better solution

IS-N-TH-POW(x)

```
 1  for exp ← 2 to ⌊log_2(n)⌋
 2        do k ← ⌈n/2⌉
 3           k_max ← n
 4           k_min ← 0
 5           while k ≠ k_max
 6                 do test ← k^exp
 7                    if test = n
 8                       then return true
 9                    if test > n
10                       then k_max ← k
11                       else  k_min ← k
12                    k ← ⌈(k_max + k_min)/2⌉
13  return false
```

Is this a perfect power?

Complexity

- The outer `for` loop runs $\log_2(n) - 1$ times by construction
- The inner `while` is slightly more complex to analyze: examine the evolution of the k_{min} and k_{max} variables
- The viable range for the value of k is cut into a half at each loop body execution, k_{max} retains the upper bound
- Since the value of k is always an integer (initial value n): $\log_2(n)$ runs are made
- The main cost of the loop body is the exponentiation: takes $\log_2(exp)$
- The total complexity is $\sum_{exp=2}^{\log_2(n)} \log_2(n) \log(exp) = \log_2(n) \sum_{exp=2}^{\log_2(n)} \log(exp) = O(\log^2(n) \log(\log(n)))$

Multiple Precision Multiplications

The Roman Way

- Multiplying two n-digit numbers $a = (a_{n-1}, \ldots, a_0)_t$, $b = (b_{n-1}, \ldots, b_0)_t$ represented in base t (think 10, or 2) was a problem in ancient Rome, as they did it like this:

MULTIPLY(a, b)

1 $res \leftarrow (0_{2n-1}, 0_{2n-2}, \ldots, 0_1, 0_0)$ $\triangleright$ res can be up to $2n$ digits
2 **for** $i \leftarrow 0$ **to** $b - 1$
3 **do** $res \leftarrow$ ADDWITHCARRY(res, a)
4 **return** res

- The call to ADDWITHCARRY needs to take care of all the possible carries, thus takes $O(n)$. Total complexity?
- The for loop runs as many times as the size of b: Worst case, b is n-digits in base t, thus $b \approx t^n$: $O(nt^n)$

Back to first grade school

Multiple Digits Multiplications

- We were taught to go faster than ancient Romans in first grade school, namely we do :

MULTIPLY(a, b)

1 $res \leftarrow (0_{2n-1}, 0_{2n-2}, \ldots, 0_1, 0_0)$
2 **for** $i \leftarrow 0$ **to** $n-1$
3 **do** $tmp = 0$
4 **for** $j \leftarrow 0$ **to** $n-1$
5 **do** $prod \leftarrow a_j * b_i$
6 $tmp \leftarrow tmp + (prod \times t^{i+j})$
7 $res \leftarrow$ ADDWITHCARRY(res, tmp)
8 **return** res

Back to first grade school

Time Complexity for Schoolbook Multiplication

- Each iteration of the inner loop executes in constant time:
  - *prod* is at most 2 digits
  - Line 6 sums *prod* to the 2 digits tmp_{i+j}, tmp_{i+j+1}
  - No carries generated as $tmp_{i+j+1} = 0$
- The inner loop runs n times, thus it costs $O(n)$
- Each iteration of the outer loop requires $O(n)$ for the inner loop and $O(n)$ for AddWithCarry $\Rightarrow O(n)$complexity
- The outer loop runs n times each, the loop body is $O(n)$, thus $O(n \times n) = O(n^2)$ Better!

Similarly for multiple digits divisions:

- Division with repeated subtractions is $O(nt^n)$.
- Schoolbook division is $O(n^2)$

Improving Multiplication

Karatsuba's algorithm

KARATSUBAMULT(a, b, n)

1 $\triangleright$ a, b are integers in base t with n digits
2 **if** $n = 1$
3 **then return** $a * b$
4 $m \leftarrow \lfloor \frac{n}{2} \rfloor$
5 $\triangleright$ Shift operations are performed over digits of the integers
6 $high_a \leftarrow a >> m$
7 $low_a \leftarrow a - (high_a << m)$
8 $high_b \leftarrow b >> m$
9 $low_b \leftarrow b - (high_b << m)$
10 $z_0 \leftarrow$ KARATSUBAMULT(low_a, low_b, m)
11 $z_1 \leftarrow$ KARATSUBAMULT$(high_a + low_a, high_b + low_b, m + 1)$
12 $z_2 \leftarrow$ KARATSUBAMULT$(high_a, high_b, m)$
13 **return** $z_2 << 2m + (z_1 - z_2 - z_0) << m + z_0$

Improving Multiplication

Why Does It Work?

- $a = high_a \cdot t^m + low_a$ and $b = high_b \cdot t^m + low_b$
- $a \cdot b = high_a \cdot t^m \cdot high_b \cdot t^m + low_a \cdot high_b \cdot t^m + high_a \cdot low_b \cdot t^m + low_a \cdot low_b = z_2 \cdot t^{2m} + t^m \cdot (low_a \cdot high_b + high_a \cdot low_b) + z_0$
- Now, we show that $high_a \cdot low_b + low_a \cdot high_b = z_1 - z_2 - z_0$: $z_1 = (high_a + low_a) \cdot (high_b + low_b) = high_a \cdot high_b + high_a \cdot low_b + low_a \cdot high_b + low_a \cdot low_b = z_2 + high_a \cdot low_b + low_a \cdot high_b + z_0$
- $a \cdot b = z_2 \cdot t^{2m} + t^m \cdot (low_a \cdot high_b + high_a \cdot low_b) + z_0 = z_2 \cdot t^{2m} + t^m \cdot (z_1 - z_2 - z_0) + z_0$
- We can perform the multiplication with 3 multiplications on half digits numbers, plus some shift/additions, until we get a single digit multiplication, which stops the recursion
- Basically, only single digit multiplications are performed and the results are composed using the equation in line 13

Improving Multiplication

Complexity

- The algorithm is recursive!
- Recursion always involve sub-problems, which has the same complexity function of the main problem
- Thus, we can formulate the time complexity as a function of the costs of the sub-problems, defining the so-called recurrence equation
- How many sub-problems? 3
- What size of these sub-problems? $\lceil \frac{n}{2} \rceil$
- What is the cost of the function without considering the sub-problems? $\Theta(n)$, since only shifts and additions are performed
- Thus, $T(n) = 3T(\lceil \frac{n}{2} \rceil) + \Theta(n)$

Analyzing recursions

How to solve recursions?

- The previous time complexity is also stated in a recurrent fashion: we need to solve the issue
- There are two main ways to do so:
  - Master's theorem: provided the recurrence fits certain conditions, easy, closed form solutions are available
  - Hypothesize and prove by induction: we obtain an educated guess on the total complexity through partially unrolling the recursion, and subsequently prove that our conjecture is correct by means of mathematical induction

Master's Theorem

A smart toolbox

- The master's theorem provides a ready-made toolbox for the complexity analysis of recursions.
- The recursion complexity must be expressible in this form : $T(n) = aT(\frac{n}{b}) + f(n)$ with $a \geq 1, b > 1$
- Key idea: compare the complexities of $n^{log_b(a)}$ (the effect of the recursive calls) and $f(n)$ (the cost of running the function)
- Hypotheses for the Master's theorem:
  - a must be constant, and greater or equal to one (at least one sub-problem per recursion)
  - $f(n)$ must be added, not subtracted or anything else to $aT(\frac{n}{b})$
  - The difference between $n^{\log_b(a)}$ and $f(n)$ should be polynomial
- The solutions provided by the MT are split into three cases

Master's Theorem

Case 1

- Case 1 : $f(n) = O(n^{log_b(a)-\epsilon})$ for some $\epsilon > 0$
- Result : $T(n) = \Theta(n^{log_b(a)})$
- Key idea : the recursion dominates the complexity
- Example : $T(n) = 32\,T(\frac{n}{4}) + n\log(n)$
- Compare : $O(n^{\log_4(32)-\epsilon}) = n\log(n) \Rightarrow O(n^{\frac{5}{2}-\epsilon}) = n\log(n) \Rightarrow \epsilon = \frac{1}{2}$ ✓
- The complexity of the example is $\Theta(n^{\log_4(32)}) = \Theta(n^{\frac{5}{2}})$

Master's Theorem

Case 2

- Case 2 : $f(n) = \Theta(n^{\log_b(a)})$
- Result : $T(n) = \Theta(n^{\log_b(a)} \log(n))$
- Key idea : the "weights" of the two parts of the sum are the same up to a polynomial term
- Example: $T(n) = T(\frac{n}{3}) + \Theta(1)$
- Compare: $\Theta(1) = \Theta(n^{\log_3(1)})$?
  - Yes : $\Theta(1) = \Theta(n^0)$ ✓
- The complexity of the example is $\Theta(n^{log_3(1)} \log(n)) = \Theta(log(n))$

Master's Theorem

Case 3

- Case 3 : $f(n) = \Omega(n^{log_b(a)+\epsilon})$, $\epsilon > 0$
- In this case, for the theorem to hold, we need also to check that : $af(\frac{n}{b}) \leq cf(n)$ for some $c < 1$
- Result : $T(n) = \Theta(f(n))$
- Key idea : the function call at each recursion step outweights the rest
- Example $T(n) = 2T(\frac{n}{4}) + n\log(n)$
- Compare $n\log(n) = \Omega(n^{\log_4(2)+\epsilon}) \Rightarrow n\log(n) = \Omega(n^{\frac{1}{2}+\epsilon}) \Rightarrow \epsilon = \frac{1}{2} > 0$ ✓
- Check $2f(\frac{n}{4}) = 2\frac{n}{4}\log(\frac{n}{4}) \leq cn\log(n)$ for $c < 1$?
  - $2\frac{n}{4}\log(\frac{n}{4}) = \frac{n}{2}\log(n) - \frac{n}{2}\log(4) < \frac{n}{2}\log(n) \leq cn\log(n)$ for $c \in [\frac{1}{2}; 1)$ ✓
- The complexity of the example is $\Theta(n\log(n))$

Improving Multiplication

Karatsuba Algorithm

- Let's go back to Karatsuba's recurrence:
  $T(n) = 3T(\lceil \frac{n}{2} \rceil) + \Theta(n)$
- Can we solve it through master theorem?
- Yes! $a = 3, b = 2$ and $f(n) = \Theta(n)$
- Which case?
- $n^{\log_b(a)} = n^{\log_2(3)}$.
  $O(n^{\log_2(3)-\epsilon}) = n \Rightarrow \log_2(3) - \epsilon >= 1 \Rightarrow \epsilon = \log_2(3) - 1 > 0$,
  thus it is case 1 of the theorem
- Therefore, $T(n) = \Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(3)}) = \Theta(n^{1.585})$

When being a Master is not enough...

Alternatives to Master's Theorem

- Some cases where the MT is not applicable are:
  1. $T(n) = 2T(n-3) + \Theta(\log(n))$
     - $b \not> 1$, the MT needs $b > 1$
  2. $T(n) = 3T(\log(n)) + 2^n$
     - The recursion argument is not polynomial (so , for sure $b \not> 1$)
  3. $T(n) = nT(n-2) + \Theta(1)$
     - a is not constant ($a = n$ in this case)
- We will now solve them via inductive proofs

Solutions

Example 1 - Conjecture

- Let's try expanding a step of the recursion $T(n) = 2T(n-3) + \Theta(\log(n))$
- $T(n) = 2(2T(n-6) + \Theta(\log(n-3))) + \Theta(\log(n))$
- Each recursion step adds a term which is $2^i \log(n-3i))$
- The number of recursive calls is $\frac{n}{3}$ (remember division by iterated subtraction?)
- The expanded equation is $O(\sum_{i=0}^{\frac{n}{3}} 2^i \log(n-3i))$
- We thus conjecture that $T(n)$ is $O(2^{\frac{n}{3}} \log(n))$

Solutions

Example 1 - Proof

- We want to prove that $T(n)$ is $O(2^{\frac{n}{3}} \log(n))$, i.e. $T(n) \leq c2^{\frac{n}{3}} \log(n)$
- By induction, assume $T(n-3) \leq c2^{\frac{n-3}{3}} \log(n-3)$ (this is a reasonable induction hp, as clearly $(n-3) < n$)
- Substitute the previous one in the original recurrence equation $T(n) = 2T(n-3) + d\log(n)$, yielding $T(n) = 2T(n-3) + d\log(n) \leq c2 \cdot 2^{\frac{n-3}{3}} \log(n-3) + d\log(n)$
- Manipulating : $T(n) \leq c2 \cdot 2^{-1} \cdot 2^{\frac{n}{3}} \log(n-3) + d\log(n)$
- As clearly $\log(n-3) < \log(n)$, we get to: $T(n) \leq c2^{\frac{n}{3}} \log(n) + d\log(n)$
- Can we drop $d\log(n)$?

Solutions

Example 1 - Proof (Continued)

We can drop $\log(n)$ with a trick:

- Let's change our thesis to be proven to:
  $T(n) \leq c2^{\frac{n}{3}} \log(n) - bn - a \in O(2^{\frac{n}{3}} \log(n))$
- Induction hypothesis becomes:
  $T(n-3) \leq c2^{\frac{n-3}{3}} \log(n-3) - b(n-3) - a$
- $T(n) = 2T(n-3) + d\log(n) \leq 2c2^{\frac{n-3}{3}} \log(n-3) - 2b(n-3) - 2a + d\log(n) = 2 \cdot 2^{-1} c2^{\frac{n}{3}} \log(n-3) - 2bn + 6b - 2a + d\log(n) \leq c2^{\frac{n}{3}} \log(n) - 2bn + 6b - 2a + dn$
- Now, if $d - 2b \leq -b$ and $6b - 2a \leq -a$, we get:
  $c2^{\frac{n}{3}} \log(n) - 2bn + dn + 6b - 2a \leq c2^{\frac{n}{3}} \log(n) - bn - a$
- As a conclusion, $T(n) \leq c2^{\frac{n}{3}} \log(n) - bn - a$, if
  $d - 2b \leq -b \Rightarrow b \geq d$ and $6b - 2a \leq -a \Rightarrow a \geq 6b$

Solution

Example 1 - Smaller Upper Bound?

Let's look at the proof: Do we really exploit the $\log(n)$ term multiplied to $2^{\frac{n}{3}}$?

- Maybe $T(n) \in O(2^{\frac{n}{3}})$. Let's try to prove it!
- Thesis: $T(n) \leq c2^{\frac{n}{3}} - bn - a \in O(2^{\frac{n}{3}})$
- Induction hypothesis: $T(n-3) \leq c2^{\frac{n-3}{3}} - b(n-3) - a$
- $T(n) = 2T(n-3) + d\log(n) \leq$
  $2c2^{\frac{n-3}{3}} - 2b(n-3) - 2a + d\log(n) =$
  $2 \cdot 2^{-1}c2^{\frac{n}{3}} - 2bn + 6b - 2a + d\log(n) \leq c2^{\frac{n}{3}} - 2bn + 6b - 2a + dn$
- As in the previous proof, eventually:
  $T(n) \leq c2^{\frac{n}{3}} - 2bn + 6b - 2a + dn \leq c2^{\frac{n}{3}} - bn - a$, if
  $d - 2b \leq -b \Rightarrow b \geq d$ and $6b - 2a \leq -a \Rightarrow a \geq 6b$

$$T(n) \in O(2^{\frac{n}{3}})!$$

Solutions

Example 2 - Conjecture

- Start by expand a step of $T(n) = 3T(\log(n)) + 2^n$
- $T(n) = 3(3T(\log(\log(n))) + 2^{log(n)}) + 2^n$
- At each recursion step, the added contribution is a term of lower order w.r.t. the ones already present
- This time we can already stop the analysis now and say that $T(n) = O(2^n)$ as all other added terms are negligible w.r.t. 2^n
- More precisely, if the number of steps of the recursion is a function $f(n)$, then $T(n) = \Theta(\sum_{i=0}^{f(n)} 3^i 2^{log^i(n)})$, where $\log^i$ denotes the logarithm function repeatedly applied i times
- $f(n)$ is surely less than $\log(n)$ as we have $\log(n)$ recursive step when the size of the problem is simply halved, thus $T(n) = O(\sum_{i=0}^{\log(n)} 3^i 2^{\log^i(n)}) = O(2^n)$

Solutions

Example 2 - Proof

- Thesis: $T(n) \leq c2^n \in O(2^n)$
- Induction Hypothesis: $T(\log(n)) \leq c2^{\log(n)} = cn^{\log(2)} \leq cn$
- $T(n) = 3T(\log(n)) + 2^n \leq 3cn + 2^n \leq 3c2^n + 2^n$
- If $3c + 1 \leq c$, we can get to our desired result
- But $3c + 1 \leq c \iff c \leq \frac{-1}{2}$, but $c > 0$
- Let's try another transformation: $T(n) \leq 3cn + 2^n \leq \frac{c}{2}2^n + 2^n$
- This is legitimate since there exists $n_0 = 5$ such that $\forall n \geq n_0(\frac{c}{2}2^n \geq 3cn)$
- Now, $\frac{c}{2} + 1 \leq c \iff c \geq 2$, which is ok
- Hence, we can do: $T(n) \leq \frac{c}{2}2^n + 2^n \leq c2^n$

Solutions

Example 3 - Conjecture

- Again, expand a step of $T(n) = nT(n-2) + \Theta(1)$
- $T(n) = n((n-2)T(n-4) + \Theta(1)) + \Theta(1) =$
  $(n^2 - 2n)T(n-4) + \Theta(n) + \Theta(1) =$
  $(n^2 - 2n)((n-4)T(n-6) + \Theta(1)) + \Theta(n) + \Theta(1) =$
  $(n^3 - 6n^2 + 8n)T(n-6) + \Theta(n^2) + \Theta(n) + \Theta(1)$
- At every recursion step, a term polynomially greater than the previous is added.
- The recursion is $\frac{n}{2}$ steps deep
- The dominating complexity is the one added at the last step
- As each step raises the polynomial complexity by one, the final complexity is $O(n^{\frac{n}{2}})$

Solutions

Example 3 - Proof

- Again, proof by induction, assume $T(n-2) \leq c(n-2)^{\frac{(n-2)}{2}}$
- Substitute and obtain
  $T(n) = nT(n-2) + \Theta(1) \leq n(c(n-2)^{\frac{(n-2)}{2}}) + d$
- Manipulate a bit and get $T(n) \leq cn(n-2)^{\frac{(n-2)}{2}} + d$
- Note that $cn(n-2)^{\frac{(n-2)}{2}} < cnn^{\frac{(n-2)}{2}} = cn^{\frac{n}{2}}$
- Plug it in the previous inequality and obtain, $T(n) \leq cn^{\frac{n}{2}} + d$
- Change the thesis to get rid of d: $T(n) \leq cn^{\frac{n}{2}} - b$
- $T(n) \leq cn(n-2)^{\frac{n-2}{2}} - bn + d \leq cn^{\frac{n}{2}} - bn + d \leq cn^{\frac{n}{2}} - 2b + d$, since $-bn \leq -2b$ for $n \geq n_0 = 2$
- If $-2b + d \leq -b \Leftrightarrow b \geq d$, then
  $T(n) \leq cn^{\frac{n}{2}} - 2b + d \leq cn^{\frac{n}{2}} - b$

Naive Sorting

Simple idea for a sorting algorithm: Compute the minimum of the array, swap it with the first element of the array, then do the same on the sub-array starting from the second element:

Naive Sorting Algorithm

SELECTIONSORT$(a, start)$

```
 1  min ← a[start]
 2  i_min ← start
 3  for j ← start + 1 to LENGTH(a)
 4        do if a[j] < min
 5              then min ← a[j]
 6                   i_min ← j
 7  a[i_min] ← a[start]
 8  a[start] ← min
 9  if start + 1 < LENGTH(a)
10     then SELECTIONSORT(a, start + 1)
```

Naive Sorting

Complexity: Recurrence Equation

- The algorithm is recursive
- Each time a sub-array of size $n-1$ is considered $\Rightarrow T(n-1)$
- 1 recursive call
- Each recursion step scan the sub-array $\Rightarrow \Theta(n)$
- Thus, recurrence equation is $T(n) = T(n-1) + \Theta(n)$

Complexity: Determining $T(n)$

- Can we employ master theorem? No, since $b = 0$
- Let's make a guess on $T(n)$
- How many recursive calls? $\Rightarrow n$
- Each of them adds to the time complexity $n - i$
- Thus, $T(n) = \sum_{i=0}^{n-1} n - i = \frac{n(n-1)}{2} = O(n^2)$

Naive Sorting

Proof

- Again, our thesis is $T(n) \leq cn^2$
- Induction hypothesis: $T(n-1) \leq c(n-1)^2$
- $T(n) = T(n-1) + kn \leq c(n-1)^2 + kn =$
  $cn^2 + c - 2cn + kn \leq cn^2 + cn - 2cn + kn = cn^2 - cn + kn$
- If $k - c \leq 0 \Rightarrow c \geq k$, then we got:
  $T(n) \leq cn^2 - cn + kn \leq cn^2$

Is $T(n) \in O(n \log(n))$?

- Now, our thesis is $T(n) \leq cn \log(n)$
- Induction hypothesis: $T(n-1) \leq c(n-1) \log(n-1)$
- $T(n) \leq c(n-1) \log(n-1) + kn \leq cn \log(n) + kn$
- Can we get rid of kn?

Naive Sorting

- Let's try the usual trick: $T(n) \leq cn\log(n) - bn$
- Induction hp: $T(n-1) \leq c(n-1)\log(n-1) - b(n-1)$
- $T(n) \leq c(n-1)\log(n-1) - bn + b + kn \leq cn\log(n) - bn + b + kn$
- To get $-bn$, we need to have $k - b < -b$, which holds for $k < 0$, but $k > 0 \Rightarrow$ impossible
- Idea: we may try to introduce a new linear term $hn \geq b$: $T(n) \leq cn\log(n) - bn + hn + kn + b$, for some $h > 0$.
- To get $-bn$, we need to have $k - b + h < -b \Rightarrow k + h < 0$, which has no solution since $k, h > 0$
- Can we avoid getting rid of $n-1$ and using the term $-c\log(n)$ to get another linear term?
  1. $-c\log(n) \leq -cn$ does not hold for $c > 0$
  2. $-c\log(n) \leq cn$ is ok, but we would get $c + k < 0$, which has still no solution

Naive Sorting

In conclusion, we cannot get rid of the kn term in the recurrence!

- This is the reason why we do not erase terms with lower order in these proofs
- In this case, linear terms summed up to make the complexity $\Omega(n^2)$

Proving Lower Bound

- Let's prove that $T(n) = \Omega(n^2)$, that is $T(n) \geq cn^2, c > 0$
- Inductive hypothesis: $T(n-1) \geq c(n-1)^2 = cn^2 + c - 2cn$
- Proof: $T(n) = T(n-1) + kn \geq cn^2 + c - 2cn + kn > cn^2 - 2cn + kn \geq cn^2$ if $-2c + k \geq 0$
- In conclusion, $T(n) \geq cn^2$ if $-2c + k \geq 0 \iff c \leq \frac{k}{2}$

Differences on Proofs

Consider the following recurrence equations:

- $T(n) = 2T(\frac{n}{2}) + n$
- $T(n) = T(\frac{n}{2}) + n$

Are they $O(n)$?

Equation 1

- $T(n) \leq 2c\frac{n}{2} + n \leq cn + n$
- We cannot get rid of n, since $c + 1 > c \forall c$
- Indeed, this is the merge sort recurrence, which is $\Omega(n \log(n))$

Equation 2

- $T(n) \leq c\frac{n}{2} + n = \frac{c}{2}n + n$
- Now, if we impose $\frac{c}{2} + 1 \leq c \Rightarrow c \geq 2$, we can state $\frac{c}{2}n + n \leq cn$, thus $T(n) \leq cn$

Induction Proofs: Summing Up

Recap of all the Tricks in Proofs

- Change the thesis to try to get rid of lower order terms $\Rightarrow$ $T(n) \leq cn$ becomes $T(n) \leq cn - b$
- Play with constants $\Rightarrow$ Suppose our thesis is $T(n) \leq cn$, and we got to $T(n) \leq \frac{c}{2}n$. We can get to thesis since $\frac{c}{2} \leq c$
- If a term is added, you can replace it with an higher order term and an arbitrary constant (usually we want to decrease the constant factor) $\Rightarrow$ $T(n) \leq cn + 2^n \leq \frac{c}{2}2^n + 2^n \leq c2^n$, for $c \geq 2$
- If a term is subtracted, you can replace it with a lower order term and an arbitrary constant (usually we want to increase the constant factor) $\Rightarrow$
  $T(n) \leq cn^2 - bn + k \leq cn^2 - 2b + k \leq cn^2 - b$, for $b \geq k$

Greatest Common Divisor

- Compute the greatest common divisor between 2 integers a, b
- Suppose, as a precondition, $a \geq b$
- Naive idea: Try all possible integers starting from b to 1

The Algorithm

$\mathrm{GCD}(a, b)$

```
1  for i ← b downto 1
2        do if a mod i = 0 ∧ b mod i = 0
3              then return i
```

Complexity: the loop runs at most b times, one for each candidate divisor $\Rightarrow O(b)$

Greatest Common Divisor

A clever solution: Euclidean algorithm!

- Main idea: $gcd(a, b) = gcd(b, a \mod b)$. Since $a \mod b$ is less than b, we reduce the size of the numbers, eventually getting to multiples.

Euclidean Algorithm

EuclideanGCD(a, b)

```
1  r ← a mod b
2  n ← b
3  d ← b
4  while r ≠ 0
5        do d ← r
6           r ← n mod d
7           n ← d
8  return d
```

Greatest Common Divisor

Complexity

- The time complexity is surely dominated by the loop
- How many times does it run?
- Different cases:
  - If $gcd(a, b) = b$, the algorithm does not even get into the loop
  - There may be two steps even if $gcd(a, b)$ is 1 (e.g. $gcd(32, 15)$)
  - There may be a lot of steps: $gcd(21, 13)$
- Let's try to identify the worst case
- The maximum number of iterations is achieved if the remainder is close to d
- Turns out that such worst case is when we want to compute $gcd(f_{n+1}, f_n)$, where f_n is the $n - th$ element of Fibonacci's sequence!

Greatest Common Divisor

Worst Case Complexity

- Indeed, $f_{n+1} < 2f_n$, thus $r = f_{n+1} - f_n$, which is f_{n-1}, that is a Fibonacci number itself, which is quite big
- Most importantly, the next step of the algorithm will compute $gcd(f_n, f_{n-1})$, which still exhibit the same behavior
- Therefore, we have a sequence of r values, which are the Fibonacci's sums, starting from $f_{n-1} = a - b$ to $f_1 = 1$
- Thus, we have $O(n)$ iterations, but what is n?
- hint: $f_n \approx \phi^n$, where $\phi = \frac{1+\sqrt{5}}{2}$, the golden ratio!
- Indeed, it is easy to prove it by induction: $f_n = f_{n-1} + f_{n-2} \ldots$
- Since, $b = f_n$, then $n = \log_\phi(b)$
- In conclusion, worst case complexity is $O(n) = O(\log(b))$