Tirgul 2

Asymptotic Analysis

Asymptotic Analysis

- Motivation: Suppose you want to evaluate two programs according to their run-time for inputs of size n. The first has run-time of:

$$0.1 \cdot n^4 + \log n + 7$$

and the second has run-time of:

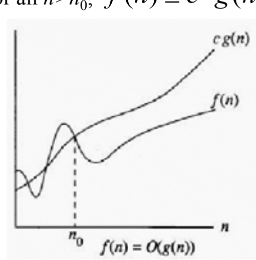
$$1000 \cdot n + 200\sqrt{n} + (\log n + 239)^2 + 3859$$

For small inputs, it doesn't matter, both programs will finish before you notice. What about (really) large inputs?

Big- O

- **Definition:**

$f(n) = O(g(n))$ if there exist constants c>0 and n_0 such that for all n>n_0, $f(n) \le c \cdot g(n)$



Big- O

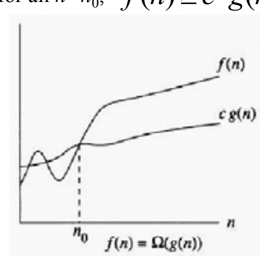
- In other words, $g(n)$ bounds $f(n)$ from above (for large n's) up to a constant.
- Examples:

1) $1000000 = O(1)$
2) $0.5n = O(n)$
3) $10000\ n = O(n)$
4) $n = O(n^2)$
5) $n^2 \ne O(n)$ (why?)

Big- Omega

- **Definition:**

$f(n) = \Omega(g(n))$ if there exist constants c>0 and n_0 such that for all n>n_0, $f(n) \ge c \cdot g(n)$



Big- Omega

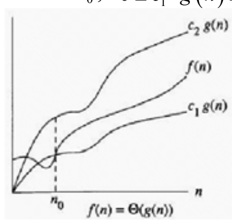
- In other words, $g(n)$ bounds $f(n)$ from below (for large n's) up to a constant.
- Examples:

1) $0.5n = \Omega(n)$
2) $10000\ n = \Omega(n)$
3) $n^2 = \Omega(n)$
4) $n \ne \Omega(n^2)$

Big- Theta

- **Definition:** $f(n) = \Theta(g(n))$ if:

  $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- This means there exist constants $c_1 > 0$, $c_2 > 0$ and n_0 such that for all $n > n_0$, $0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$



Example 1
(question 2-4-e. in Cormen)

Question: is the following claim true?

Claim: If $f(n) \ge \alpha > 0$ (for $n>n_0$) then

$$f(n) = O((f(n))^2)$$

Answer: Yes.

Proof: Take $c = 1/\alpha$. Thus for $n>n_0$,

$$f(n) = \frac{1}{\alpha} \cdot \alpha \cdot f(n) \le \frac{1}{\alpha} \cdot f(n) \cdot f(n) = c \cdot (f(n))^2$$

Big- Theta

- In other words, $g(n)$ is a tight estimate of $f(n)$ (in asymptotic terms).
- Examples:

$$1)\quad 0.5n = \Theta(n)$$
$$2)\quad n^2 \ne \Theta(n)$$
$$3)\quad n \ne \Theta(n^2)$$

Example 2
(question 2-4-d. in Cormen)

Does $f(n) = O(g(n))$ imply $2^{f(n)} = O(2^{g(n)})$?

Answer: No.

Proof : Look at, $f(n) = 2n$, $g(n)=n$,

Clearly $f(n)=O(g(n))$ (look at $c=2$ $n_0=1$).

However, given c and n_0, choose n for which $n > n_0$ and $2^n > c$, and then :

$f(n) = 2^{2n} = 2^n * 2^n > c * 2^n = c * g(n)$

Example 1

Question: is the following claim true?

Claim: For all f, (for large enough n, i.e. $n > n_0$)

$$f(n) = O((f(n))^2)$$

Answer : No.

Proof : Look at $f(n) = 1/n$.

Given c and n_0, choose n large enough so $n>n_0$ and $1/n < c$. For this n, it holds that

$(f(n))^2 = 1/n^2 = 1/n * 1/n < c * 1/n. = c*f(n)$

Summations
(from Cormen, ex. 3.2-2., page 52)

Find the asymptotic upper bound of the sum $\sum_{k=0}^{\lfloor \log n \rfloor} \lceil n/2^k \rceil$

$(\lceil n/1 \rceil + \lceil n/2 \rceil + \lceil n/4 \rceil + \lceil n/8 \rceil + \ldots + \lceil 1 \rceil)$

$$\sum_{k=0}^{\lfloor \log n \rfloor} \lceil n/2^k \rceil \le \sum_{k=0}^{\lfloor \log n \rfloor} ((n/2^k)+1) \le \sum_{k=0}^{\lfloor \log n \rfloor} 1 + \sum_{k=0}^{\lfloor \log n \rfloor} n/2^k \le$$

$$\le (\log n + 1) + n \sum_{k=0}^{\infty} 1/2^k = 1 + \log n + 2n = O(n)$$

- note how we “got rid” of the integer rounding
- The first term is n so the sum is also $\Omega(n)$
- **Note that** the largest item dominates the growth of the term in an exponential decrease/increase.

Summations (example 2)

(Cormen, ex. 3.1-a., page 52)

- Find an asymptotic upper bound for the following expression:
- $f(n) = \sum_{k=1}^{n} k^r$ (r is a constant) :

$$f(n) = 1^r + 2^r + \ldots + n^r \le n \cdot n^r = n^{r+1} = O(n^{r+1})$$

note that $n \cdot n^r \neq O(n^r)$

- Note that when a series increases polynomially the upper bound would be the last element but with an exponent increased by one.
- Is this bound tight?

Example 2 (Cont.)

To prove a tight bound we should prove a lower bound that equals the upper bound.

Watch the amazing upper half trick :

Assume first, that n is even (i.e. n/2 is an integer)

$f(n) = 1^r+2^r+....+n^r > (n/2)^r+...+n^r > (n/2)(n/2)^r =$

$(1/2)^{r+1} * n^{r+1} = c * n^{r+1} = \Omega(n^{r+1})$

Technicality : n is not necessarily even.

$f(n) = 1^r+2^r+....+n^r > \lceil n/2 \rceil + \ldots + n^r \geq (n-1)/2 * (n/2)$^r

$\geq (n/2)^{r+1} = \Omega(n^{r+1})$.

Example 2 (Cont.)

- Thus: $f(n) = \Theta(n^{r+1})$ so our upper bound was tight!

Recurrences – Towers of Hanoi

- The input of the problem is: s, t, m, k
- The size of the input is k+3 ~ k (the number of disks).
- Denote the size of the problem k=n.
- Reminder:

```
H(s,t,m,k) {
/* s - source, t - target, m - middle */
  if (k > 1) {
    H(s,m,t,k-1)
      /* note the change, we move from the
         source to the middle */
    moveDisk(s,t)
    H(m,t,s,k-1)
  } else { moveDisk(s,t) }
}
```

- What is the running time of the "Towers of Hanoi"?

Recurrences

- Denote the run time of a recursive call to input with size n as $h(n)$
- H(s, m, t, k-1) takes $h(k-1)$ time
- moveDisk(s, t) takes $h(1)$ time
- H(m, t, s, k-1) takes $h(k-1)$ time
- We can express the running-time as a recurrence:

$$h(n) = 2h(n-1) + 1$$
$$h(1) = 1$$

- How do we solve this ?
- A method to solve recurrence is **guess** and prove by **induction.**

Step 1: "guessing" the solution

$h(n) = 2h(n-1) + 1$

$= 2[2h(n-2)+1] + 1 = 4h(n-2) + 3 = 2^2 h(n-2) + 2^2 - 1$

$= 4[2h(n-3)+1] + 3 = 8h(n-3) + 7 = 2^3 h(n-3) + 2^3 - 1$

- When repeating k times we get:

$$h(n)=2^k h(n-k) + (2^k - 1)$$

- Now take k=n-1. We'll get:

$h(n) = 2^{n-1} h(n-(n-1)) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1$
$= 2^n - 1$

Step 2: proving by induction

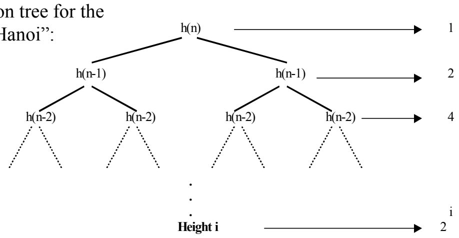
- If we guessed right, it will be easy to prove by induction that $h(n)=2^n - 1$
- For n=1 : $h(1)= 2\text{-}1=1$ (and indeed $h(1)=1$)
- Suppose $h(n\text{-}1) = 2^{n-1} - 1$. Then,

$$h(n) = 2h(n-1) + 1 = 2(2^{n-1} - 1) + 1$$
$$= 2^n -2 + 1 = 2^n -1$$

- So we conclude that: $h(n) = O(2^n)$

Recursion Trees

The recursion tree for the "towers of Hanoi":



- For each level we write the time added due to this level. In Hanoi, each recursive call adds one operation (plus the recursion). Thus the total is: $\sum_{i=0}^{n-1} 2^i = 2^n - 1$

Another Example for Recurrence

$T(n) = 2\ T(n/2) + 1$
$T(1) = 1$

$T(n) = 2T(n/2) + 1$
$= 2\ (2T(n/4) + 1) + 1 = 4T(n/4) + 3$
$= 4\ (2T(n/8) + 1) + 3 = 8T(n/8) + 7$

- And we get: $T(n) = k\ T(n/k)+(k-1)$
  For $k=n$ we get $T(n)= n\ T(1)+n-1=2n-1$
  Now proving by induction is very simple.

Another Example for Recurrence

- Another way: "guess" right away $T(n) <= c\ n - b$ (for some b and c we don't know yet), and try to prove by induction:
- The base case:
  For $n=1$: $T(1)=c\text{-}b$, which is true when $c\text{-}b=1$
- The induction step:
  Assume $T(n/2)=c(n/2)\text{-}b$ and prove for $T(n)$.
  $T(n) <= 2\ (c(n/2) - b) + 1 = c\ n - 2b + 1 <= c\ n - b$
  (the last step is true if $b>=1$). Conclusion: $T(n) = O(n)$

Beware of common mistake!

Lets prove that $2^n=O(n)$ (This is **wrong**)

For $n=1$ it is true that $2^1 = 2 = O(1)$.

Assume true for i, we will prove for $i+1$:

$f(i+1) = 2^{i+1} = 2*2^i = 2*f(i) = 2*O(n) = O(n)$.

What went Wrong?

We can **not** use the $O(f(n))$ in the induction, the O notation is only short hand for the definition itself. We should use the definition

Beware of common mistake!(cont)

If we try the trick using the exact definition, it fails.

Assume $2^n=O(n)$ then there exists c and n_0 such that for all $n > n_0$ it holds that $2^n < c*n$.

The induction step :

$f(i+1) = 2^{i+1}=2*2^i \le 2*c*i$ but it is **not** true that

$2*c*i \le c*(i+1)$.

DAST 2004

If we have time…….

DAST 2004

The little *o(f(n))* notation

Intuitively, $f(n)=O(g(n))$ means
"$f(n)$ does not grow much faster than $g(n)$".
We would also like to have a notation for
"$f(n)$ grows slower than $g(n)$".
The notation is $f(n) = o(g(n))$.
(Note the *o* is **little** *o*).

DAST 2004

Little *o*, definition

Formally, $f(n)=O(g(n))$, iff
For every positive constant *c*, there exists an n_0
Such that for all $n > n_0$, it holds that

$$f(n) < c^* g(n).$$

For example, $n = o(n^2)$, since,
Given $c>0$, choose $n_0 > 1/c$, then for $n > n_0$
$f(n) = n = c*1/c*n < c*n_0*n < c*n^2 = c*g(n)$.

DAST 2004

Little o cont'

However,, $n \neq o(n)$, since for the constant $c=2$
There is no n_0 from which $f(n) = n > 2*n.= c*g(n)$.

Another example, $\sqrt{n} = o(n)$, since,
Given $c>0$, choose n_0 for which $\sqrt{n_0} > 1/c$,
then for $n> n_0$:
$f(n) = \sqrt{n} = c*1/c*\sqrt{n} < c*\sqrt{n_0} *\sqrt{n} < c*\sqrt{n} *\sqrt{n}$
$= c*n$