Tirgul 1

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:

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

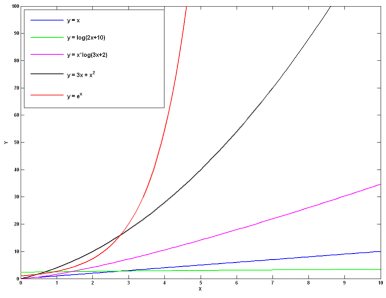
and the second has run-time of:

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

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

Asymptotic Analysis

Some of the functions you might meet:



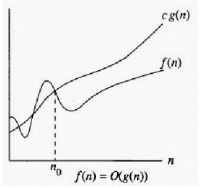
Big **O**

Definition:

$f(n) = O(g(n))$

if there exist constants $c > 0$ and n_0 such that for all $n > n_0$,

$f(n) \leq c \cdot g(n)$



Big **O**

- In other words, $g(n)$ bounds $f(n)$ from above (for large n's) up to a constant.
- Examples:
  1. 1,000,000 = $\boldsymbol{O}(1)$
  2. $0.5n$ = $\boldsymbol{O}(n)$
  3. $1{,}000n$ = $\boldsymbol{O}(n)$
  4. n = $\boldsymbol{O}(n^2)$
  5. n^2 ≠ $\boldsymbol{O}(n)$ (why?)

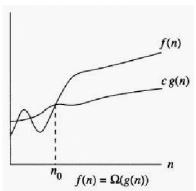
Big Omega (?)

Definition:

$f(n) = \Omega(g(n))$

if there exist constants $c>0$ and n_0 such that for all $n>n_0$,

$f(n) \geq c \cdot g(n)$



Big Omega (?)

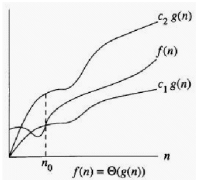
- In other words, *g(n)* bounds *f(n)* from below (for large *n*'s) up to a constant.
- Examples:
  1. $0.5n$ $= ?\ (n)$
  2. $1{,}000n$ $= ?\ (n)$
  3. n^2 $= ?\ (n)$
  4. n $\neq ?\ (n^2)$ (why?)

Big Theta (T)

Definition:

$f(n) = \Theta(g(n))$ if
$f(n) = O(g(n))$ and
$f(n) = \Omega(g(n))$

In other words, there exist constants c_1, c_2 and n_0 such that for all $n > n_0$



$$0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$$

Big Theta (T)

- In other words, *g(n)* is a tight estimate of *f(n)* (in asymptotic terms).
- Examples:
  1. $0.5n$ $= \mathbf{T}(n)$
  2. $1{,}000n$ $= \mathbf{T}(n)$
  3. n^2 $\neq \mathbf{T}(n)$
  4. n $\neq \mathbf{T}(n^2)$

Example 1

(question 2-4-e. in Cormen)

Question: is the following claim true?

Claim: If $f(n) \geq a > 0$ (for $n > n_0$) then $f(n) = O((f(n))^2)$

Answer: Yes.

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

$$f(n) = \frac{1}{a} \cdot a \cdot f(n) \leq \frac{1}{a} \cdot f(n) \cdot f(n) = c \cdot f(n)^2$$

Note: if there is no such a for $f(n)$ then the claim is not true:

take for example $f(n) = 1/n$. For **any** constant c, take $n > c$

and so: $f(n) > \frac{c}{n} f(n) = \frac{c}{n^2} = c \cdot f(n)^2$

Example 2

(question 2-4-d. in Cormen)

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

Answer: No

For example, $f(n) = 2n,\ g(n) = n$

Then, $c = 2$ and $f(n) \leq 2 \cdot g(n) \Rightarrow f(n) = O(g(n))$

but, for **any** constant c, we can choose n such that:

$$2^{2n} = 2^n \cdot 2^n > c \cdot 2^n$$

so:

$$2^{f(n)} \neq O(2^{g(n)})$$

Recurrence

- So far we only saw sequential functions:
  - we could use summation to analyze the algorithm's behavior
- What happens when we have a recursive method?
- We can no longer analyze the algorithm simply by summing the basic operations …

Recurrence – Merge sort

```
MergeSort(a) {
   l = MergeSort(left half
      of a)
   r = MergeSort(right half
      of a)
   Merge(a,l,r);
}
Merge(a,l,r) {
   Left_ix=0; right_ix=0;
   For i=1:length(l+r) do
      if …
}
```

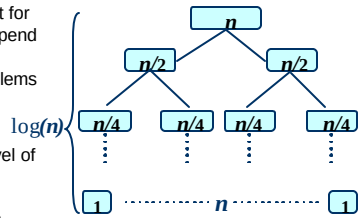
- We start with a problem of size n.
- We divide it into two sub-problems of size $n/2$ and solve them.
- We then spend some more time in constructing a solution from the partial solutions

Recurrence – Merge sort

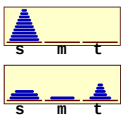
In each node, except for the leafs, we only "spend time" at splitting the problem to sub-problems and merging the solutions.

The work at each level of the tree is $O(n)$

We have $\log(n)$ such levels



Recurrence – Towers of Hanoi




```
Hanoi(s,t,m,k)
   if (k > 1) {
   H(s,m,t,k-1)
      moveDisk(s,t)
      H(m,t,s,k-1)
   } else {
      moveDisk(s,t)
   }
}
```

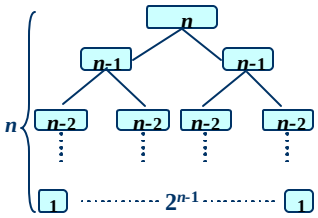
- We start with a problem of size n.
- We divide it into two sub-problems of size n-1 and solve them.
- We then spend some more time in constructing a solution from the partial solutions

Recurrence – Merge sort

In each node (including the leafs) we make a constant amount of work

Q. How much work should we do?

A 2^{n-1}



Useful to know

- When we talk of asymptotic analysis we refer to large ns.
- In "real life" the constant c (the one we used to ignore) may be important.
- For example: fast sorting algorithms use *quicksort* but when the sub-problems are small enough they use *bubblesort*.