Analysis of Algorithms, I
CSOR W4246

Eleni Drinea
Computer Science Department

Columbia University

Thursday, January 21, 2016

Outline

Review of the last lecture

- Introduced the problem of **sorting**.
- Analyzed `insertion-sort`.
  - Worst-case running time: $T(n) = \frac{3n^2}{2} + \frac{7n}{2} - 4$
  - Space: in-place algorithm
- Worst-case running time analysis: a reasonable measure of algorithmic efficiency.
- Defined polynomial-time algorithms as "efficient".
- Argued that detailed characterizations of running times are not convenient for understanding scalability of algorithms.

Running time in terms of # primitive steps

We need a coarser classification of running times of algorithms; exact characterizations

- are **too detailed**;
- do not reveal similarities between running times in an immediate way as n grows large;
- are often **meaningless**: pseudocode steps will **expand** by a constant factor that depends on the hardware.

Today

1. Asymptotic notation
2. The divide & conquer principle; application: mergesort
3. Solving recurrences and running time of `Mergesort`

Aymptotic analysis

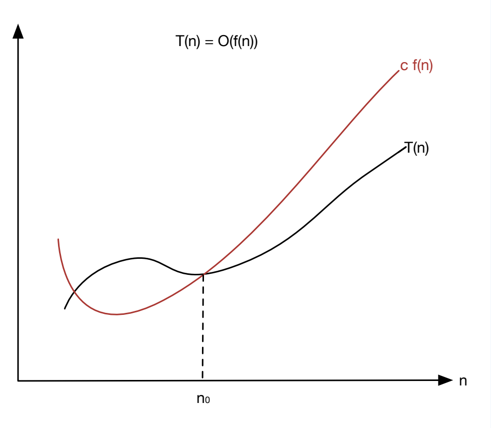
A framework that will allow us to compare the rate of growth of different running times as the input size n grows.

- We will express the running time as a function of the number of primitive steps; the latter is a function of the size of the input n.
- To compare functions expressing running times, we will ignore their low-order terms and focus solely on the highest-order term.

Asymptotic upper bounds: Big-O notation

Definition 1 (O).

We say that $T(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.



Asymptotic upper bounds: Big-O notation

Definition 1 (O).

We say that $T(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.

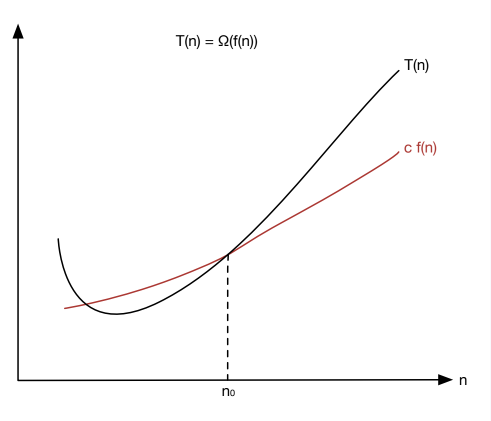
Examples:

- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^2$.
- $T(n) = an^2 + b$, $f(n) = n^3$.

Asymptotic lower bounds: Big-Ω notation

Definition 2 (Ω).

We say that $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$.



Asymptotic lower bounds: Big-Ω notation

Definition 2 (Ω).

We say that $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$.

Examples:

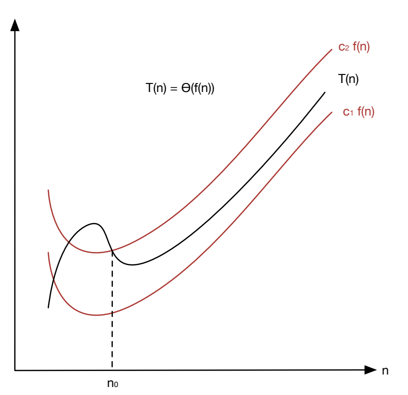
- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^2$.
- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n$.

Asymptotic tight bounds: Θ notation

Definition 3 (Θ).

We say that $T(n) = \Theta(f(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n).$$



Asymptotic tight bounds: Θ notation

Definition 3 (Θ).

We say that $T(n) = \Theta(f(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n).$$

Equivalent definition

$T(n) = \Theta(f(n)) \quad \text{if} \quad T(n) = O(f(n)) \; \textit{and} \; T(n) = \Omega(f(n))$

Asymptotic tight bounds: Θ notation

Definition 3 (Θ).

We say that $T(n) = \Theta(f(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n).$$

Equivalent definition
$T(n) = \Theta(f(n)) \quad \text{if} \quad T(n) = O(f(n)) \; \textit{and} \; T(n) = \Omega(f(n))$

Examples:

- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^2$.
- $T(n) = n \log n + n$, and $f(n) = n \log n$.

Asymptotic upper bounds that are **not** tight: little-o

Definition 4 (o).

We say that $T(n) = o(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) < c \cdot f(n)$.

Asymptotic upper bounds that are **not** tight: little-o

Definition 4 (o).

We say that $T(n) = o(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) < c \cdot f(n)$.

- Intuitively, $T(n)$ becomes insignificant relative to $f(n)$ as $n \to \infty$.
- Proof by showing that $\lim_{n\to\infty} \frac{T(n)}{f(n)} = 0$ (if the limit exists).

Asymptotic upper bounds that are **not** tight: little-o

Definition 4 (o).

We say that $T(n) = o(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) < c \cdot f(n)$.

- Intuitively, $T(n)$ becomes insignificant relative to $f(n)$ as $n \to \infty$.
- Proof by showing that $\lim_{n\to\infty} \frac{T(n)}{f(n)} = 0$ (if the limit exists).

Examples:

- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^3$.
- $T(n) = n \log n$ and $f(n) = n^2$.

Asymptotic lower bounds that are **not** tight: little-ω

Definition 5 (ω).

We say that $T(n) = \omega(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) > c \cdot f(n)$.

Asymptotic lower bounds that are **not** tight: little-ω

Definition 5 (ω).

We say that $T(n) = \omega(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) > c \cdot f(n)$.

- Intuitively $T(n)$ becomes arbitrarily large relative to $f(n)$, as $n \to \infty$.
- $T(n) = \omega(f(n))$ implies that $\lim_{n\to\infty} \frac{T(n)}{f(n)} = \infty$, if the limit exists. Then $f(n) = o(T(n))$.

Asymptotic lower bounds that are **not** tight: little-ω

Definition 5 (ω).

We say that $T(n) = \omega(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) > c \cdot f(n)$.

- Intuitively $T(n)$ becomes arbitrarily large relative to $f(n)$, as $n \to \infty$.
- $T(n) = \omega(f(n))$ implies that $\lim\limits_{n\to\infty} \frac{T(n)}{f(n)} = \infty$, if the limit exists. Then $f(n) = o(T(n))$.

Examples:

- $T(n) = n^2$ and $f(n) = n \log n$.
- $T(n) = 2^n$ and $f(n) = n^5$.

Basic rules for omitting low order terms from functions

1. Ignore **multiplicative** factors: e.g., $10n^3$ becomes n^3
2. n^a dominates n^b if $a > b$: e.g., n^2 dominates n
3. Exponentials dominate polynomials: e.g., 2^n dominates n^4
4. Polynomials dominate logarithms: e.g., n dominates $\log^3 n$

$\Rightarrow$ For large enough n,

$$\log n < n < n \log n < n^2 < 2^n < 3^n < n^n$$

Notation: $\log n$ stands for $\log_2 n$

Properties of asymptotic growth rates

1. **Transitivity**
   1.1 If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
   1.2 If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.
   1.3 If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.
2. **Sums** of (up to a constant number of) functions
   2.1 If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$.
   2.2 Let k be a fixed constant, and let $f_1, f_2, \ldots, f_k, h$ be functions such that for all i, $f_i = O(h)$. Then $f_1 + f_2 + \ldots + f_k = O(h)$.
3. **Transpose symmetry**
   - $f = O(g)$ if and only if $g = \Omega(f)$.
   - $f = o(g)$ if and only if $g = \omega(f)$.

Today

1. Asymptotic notation
2. The divide & conquer principle; application: mergesort
3. Solving recurrences and running time of `Mergesort`

The divide & conquer principle

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively.
- **Combine** the solutions to the subproblems into the solution for the original problem.

Divide & Conquer applied to sorting

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  Divide the input array into two lists of equal size.
- **Conquer** the subproblems by solving them recursively.
  Sort each list recursively. (Stop when lists have size 2.)
- **Combine** the solutions to the subproblems into the solution for the original problem.
  Merge the two sorted lists and output the sorted array.

Mergesort: pseudocode

```
Mergesort (A, left, right)
    if right == left then return
    end if
    middle = left + ⌊(right − left)/2⌋
    Mergesort (A, left, mid)
    Mergesort (A, mid + 1, right)
    Merge (A, left, mid, right)
```

Remarks

- `Mergesort` is a recursive procedure *(why?)*
- Initial call: `Mergesort`$(A, 1, n)$
- Subroutine `Merge` merges two sorted lists of sizes $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$ into one sorted list of size n. *How can we accomplish this?*

Merge: intuition

Intuition: To merge two sorted lists of size $n/2$ repeatedly

- compare the two items in the front of the two lists;
- extract the smaller item and append it to the output;
- update the front of the list from which the item was extracted.

Example: $n = 8, L = \{1, 3, 5, 7\}, R = \{2, 6, 8, 10\}$

`Merge`: pseudocode

`Merge` $(A, left, right, mid)$

$L = A[left, mid]$
$R = A[mid + 1, right]$
Maintain two pointers p_L, p_R initialized to point to the first elements of L, R, respectively
while both lists are nonempty **do**
    Let x, y be the elements pointed to by p_L, p_R
    Compare x, y and append the smaller to the output
    Advance the pointer in the list with the smaller of x, y
end while
Append the remainder of the non-empty list to the output.

Remark: the output is stored directly in $A[left, right]$, thus the subarray $A[left, right]$ is sorted after `Merge`$(A, left, right, mid)$.

`Merge`: optional exercises

Exercise 1: write detailed pseudocode (or actual code) for `Merge`

Exercise 2: write a recursive `Merge`

Analysis of `Merge`

1. **Correctness**

2. **Running time**

3. **Space**

Analysis of `Merge`: correctness

1. **Correctness:** the smaller number in the input is $L[1]$ or $R[1]$ and it will be the first number in the output. The rest of the output is just the list obtained by $\texttt{Merge}(L, R)$ after deleting the smallest element.
2. **Running time**
3. **Space**

`Merge`: pseudocode

`Merge` $(A, left, right, mid)$

$L = A[left, mid]$ $\rightarrow$**not** a primitive computational step!
$R = A[mid+1, right]$ $\rightarrow$**not** a primitive computational step!
Maintain two pointers p_L, p_R initialized to point to the first elements of L, R, respectively
while both lists are nonempty **do**
  Let x, y be the elements pointed to by p_L, p_R
  Compare x, y and append the smaller to the output
  Advance the pointer in the list with the smaller of x, y
end while
Append the remainder of the non-empty list to the output.

Remark: the output is stored directly in $A[left, right]$, thus the subarray $A[left, right]$ is sorted after `Merge`$(A, left, right, mid)$.

Analysis of `Merge`: running time

1. **Correctness:** the smaller number in the input is $L[1]$ or $R[1]$ and it will be the first number in the output. The rest of the output is just the list obtained by $\texttt{Merge}(L, R)$ after deleting the smallest element.
2. **Running time:**
   - Suppose L, R have $n/2$ elements each
   - *How many iterations before all elements from both lists have been appended to the output?*
   - *How much work within each iteration?*
3. **Space**

Analysis of `Merge`: space

1. **Correctness:** the smaller number in the input is $L[1]$ or $R[1]$ and it will be the first number in the output. The rest of the output is just the list obtained by $\texttt{Merge}(L, R)$ *after* deleting the smallest element.
2. **Running time:**
   - L, R have $n/2$ elements each
   - *How many iterations before all elements from both lists have been appended to the output?* At most $n-1$.
   - *How much work within each iteration?* Constant.

   $\Rightarrow$ `Merge` takes $O(n)$ time to merge L, R (*why?*).
3. **Space:** extra $\Theta(n)$ space to store L, R (the output of `Merge` is stored directly in A).

`Mergesort`: optional exercise

Refresh your memory on recursive algorithms by running `Mergesort` on **input:** $1, 7, 4, 3, 5, 8, 6, 2$

Analysis of `Mergesort`

1. **Correctness**
2. **Running time**
3. **Space**

`Mergesort`: correctness

For simplicity, assume $n = 2^k$, integer $k \geq 0$. We will use induction on k.

- **Base case:** For $k = 0$, the input consists of $n = 1$ item; `Mergesort` returns the item.
- **Induction Hypothesis:** For $k > 0$, assume that `Mergesort` correctly sorts any list of size 2^k.
- **Induction Step:** We will show that `Mergesort` correctly sorts any list of size 2^{k+1}.
  - The input list is split into two lists, each of size 2^k.
  - `Mergesort` recursively calls itself on each list. By the hypothesis, when the subroutines return, each list is sorted.
  - Since `Merge` is correct, it will merge these two sorted lists into one sorted output list of size $2 \cdot 2^k$.
  - Thus `Mergesort` correctly sorts any input of size 2^{k+1}.

Running time of `Mergesort`

The running time of `Mergesort` satisfies:

$$T(n) = 2T(n/2) + cn, \text{ for } n \geq 2, \text{ constant } c > 0$$
$$T(1) = c$$

This structure is typical of recurrence relations

- an **inequality** or **equation** bounds $T(n)$ in terms of an expression involving $T(m)$ for $m < n$
- a base case generally says that $T(n)$ is constant for small constant n

Remarks

- We ignore floor and ceiling notations.
- A recurrence does **not** provide an asymptotic bound for $T(n)$: to this end, we must **solve** the recurrence.

Today

1. Asymptotic notation
2. The divide & conquer principle; application: mergesort
3. Solving recurrences and running time of `Mergesort`

Solving recurrences, method 1: recursion trees

The technique consists of three steps

1. Analyze the first few levels of the tree of recursive calls
2. Identify a pattern
3. Sum over all levels of recursion

Example: analysis of running time of `Mergesort`

$T(n) = 2T(n/2) + cn, n \geq 2$

$T(1) = c$

A general recurrence and its solution

The running times of many recursive algorithms can be expressed by the following recurrence

$$T(n) = aT(n/b) + cn^k, \text{for } a, c > 0,\ b > 1, k \geq 0$$

What is the recursion tree for this recurrence?

- a is the branching factor
- b is the factor by which the size of each subproblem shrinks

$\Rightarrow$ at level i, there are a^i subproblems, each of size n/b^i

$\Rightarrow$ each subproblem at level i requires $c(n/b^i)^k$ work

- the height of the tree is $\log_b n$ levels

$\Rightarrow$ Total work: $\sum_{i=0}^{\log_b n} a^i c(n/b^i)^k = cn^k \sum\limits_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i$

Solving recurrences, method 2: Master theorem

Theorem 6 (Master theorem).

If $T(n) = aT(\lceil n/b \rceil) + O(n^k)$ *for some constants* $a > 0$, $b > 1$, $k \geq 0$, *then*

$$T(n) = \begin{cases} O(n^{\log_b a}) & , \textit{ if } a > b^k \\ O(n^k \log n) & , \textit{ if } a = b^k \\ O(n^k) & , \textit{ if } a < b^k \end{cases}$$

Example: running time of `Mergesort`

- $T(n) = 2T(n/2) + cn$:
  $a = 2, b = 2, k = 1, b^k = 2 = a \Rightarrow T(n) = O(n \log n)$

Solving recurrences, method 3: the substitution method

The technique consists of two steps

1. Guess a bound
2. Use (strong) induction to prove that the guess is correct

(See your textbook for more details.)

Remark 1 (simple vs strong induction).

1. **Simple induction:** *the induction step at n requires that the inductive hypothesis holds at step $n-1$.*
2. **Strong induction:** *the induction step at n requires that the inductive hypothesis holds at all steps $1, 2, \ldots, n-1$.*

Exercise: show that `Mergesort` runs in time $O(n \log n)$.

What about...

1. $T(n) = 2T(n-1) + 1, T(1) = 2$

2. $T(n) = 2T^2(n-1), T(1) = 4$

3. $T(n) = T(2n/3) + T(n/3) + cn$

Coming up...

More divide & conquer algorithms, including `Quicksort`, the state-of-the art algorithm for sorting.