Algorithms for Data Science

CSOR W4246

Eleni Drinea
Computer Science Department

Columbia University

Thursday, September 4, 2014

Outline

Review of the last lecture

- Introduced the problem of sorting and analyzed insertion sort.
  - Worst-case running time: $T(n) = n^2 + n - 4$
  - Space: in-place algorithm
- Argued that worst-case running time analysis is a reasonable measure of algorithmic efficiency.
- Defined polynomial-time algorithms as "efficient".
- Argued that detailed characterizations of running times are not convenient to understand scalability of algorithms.

Today

- Asymptotic notation: a framework that will allow us to compare the **rate of growth** of different running times.
- "Divide-and-conquer" design principle
  - Application: Merge-sort
- Analysis of "divide-and-conquer" algorithms: solving recurrence relations

Asymptotic upper bounds: Big-O notation

Definition (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 (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 (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 (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 (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: $T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ **and** $T(n) = \Omega(f(n))$.

Asymptotic tight bounds: Θ notation

Definition (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: $T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ **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 notation

Definition (o)

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

- Intuitively, this notation says that $T(n)$ becomes insignificant relative to $f(n)$ as $n \to \infty$.
- We can usually prove that $T(n) = o(f(n))$ 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 notation

Definition (o)

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

- Intuitively, this notation says that $T(n)$ becomes insignificant relative to $f(n)$ as $n \to \infty$.
- We can usually prove that $T(n) = o(f(n))$ 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 $f(n) = n^3$.
- $T(n) = n \log n$, $a, b, d > 0$ constants and $f(n) = n^2$.

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

Definition (ω)

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

- Intuitively, this notation says that $T(n)$ becomes arbitrarily large relative to $f(n)$ as $n \to \infty$.
- $T(n) = \omega(f(n))$ implies that the limit (if it exists)

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = \infty,$$

  thus $f(n) = o(T(n))$.

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

Definition (ω)

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

- Intuitively, this notation says that $T(n)$ becomes arbitrarily large relative to $f(n)$ as $n \to \infty$.
- $T(n) = \omega(f(n))$ implies that the limit (if it exists)

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = \infty,$$

  thus $f(n) = o(T(n))$.

Examples:

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

Properties of asymptotic growth rates

Transitivity

1. If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
2. If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
3. If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Sums of (up to a constant number of) functions

1. If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
2. Let k be a fixed constant, and let $f_1, f_2, \ldots, f_k, h$ be functions s.t. for all i, $f_i = O(h)$. Then $f_1 + f_2 + \ldots + f_k = O(h)$.

Transpose symmetry

- $f = O(g)$ if and only if $g = \Omega(f)$.
- $f = o(g)$ if and only if $g = \omega(f)$.

Divide & Conquer

The 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.
- **Combine** the solutions to the subproblems into the solution for the original problem.
  Merge the two sorted lists and output the sorted array.

Merge-Sort: pseudocode

Merge-Sort $(A, left, right)$

if $size(A) = 1$ **then** return A

end if

$middle = left + \lfloor (right - left)/2 \rfloor$

Merge-Sort $(A, left, middle)$

Merge-Sort $(A, middle + 1, right)$

Merge $(A, left, middle, right)$

- Subroutine Merge merges two **sorted** lists of sizes $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$ into one sorted array of size n.
  How can we accomplish that?

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 lists

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

Merge: pseudocode I

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

$L = A[left, \ldots, mid]$

$R = A[mid + 1, \ldots, right]$

Maintain a *Current* pointer for each list initialized to point to the first element of every list

while both lists are nonempty **do**

Let x, y be the elements pointed to by the *Current* pointers

Compare x, y and append the smaller to the output

Advance the *Current* pointer in the list from which the smaller element was selected

end while

Once one list is empty, append the remainder of the other list to the output.

Merge: pseudocode II

```
Merge(A, left, mid, right)
   L[1, ..., mid − left + 1] = A[left, ..., mid]
   R[1, ..., right − mid] = A[mid + 1, ..., right]
   pointer1 = pointer2 = 1
   index = left
   while pointer1 <= |L| and pointer2 <= |R| do
      if L[pointer1] <= R[pointer2] then
         A[index] = L[pointer1]
         pointer1 = pointer1 + 1
      else
         A[index] = R[pointer2]
         pointer2 = pointer2 + 1
      end if
      index = index + 1
   end while
   if (pointer1 > |L|) then
      A[index, ..., right] = R[pointer2, ..., right − left]
   else if (pointer2 > |R|)
      A[index, ..., right] = L[pointer1, ..., mid − left + 1]
   end if
```

Analysis of Merge

- Correctness
- Running time
- 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 $\text{Merge}(L, R)$ after deleting the smallest element.

2. **Running time:**

3. **Space:**

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 $\text{Merge}(L, R)$ after deleting the smallest element.
2. **Running time:**
   - L, R have $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$ elements respectively
   - *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 Merge(L, R) *after* deleting the smallest element.
2. **Running time:**
   - L, R have $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$ elements respectively
   - *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 sorted output is stored directly in A).

Analysis of Merge Sort

- Correctness
- Running time
- Space

Merge-sort: 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; Merge-Sort returns the item.
- **Induction Hypothesis:** For $k > 0$, assume that Merge-Sort correctly sorts any list of size 2^k.
- **Induction Step:** We will show that Merge-Sort correctly sorts any list of size 2^{k+1}.
  - The input list is split into two lists, each of size 2^k.
  - Merge-sort 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 Merge-Sort correctly sorts any input of size 2^{k+1}.

Running time of mergesort

The running time of mergesort satisfies:

$$
\begin{aligned}
&T(n) \leq 2T(n/2) + cn, \text{ for constant } c > 0, n \geq 2 \\
&T(1) \leq c
\end{aligned}
$$

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: they do not affect asymptotic bounds.
- A recurrence does **not** provide an asymptotic bound for $T(n)$: to this end, we need to solve the recurrence so that $T(n)$ appears only on the left-hand side.

How to solve recurrences

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

  Example: running time of Merge-Sort

  $T(n) = 2T(n/2) + cn, n \geq 2, T(1) \leq c$

- Substitution method
  1. Guess a bound
  2. Use induction to prove that the guess is correct

How to solve recurrences: Master theorem

Theorem (Master Theorem)

Let $a \geq 1$, $b \geq 2$ *be integers and* $c, k > 0$ *be constants. Let* $T(n)$ *be defined over the non-negative integers by the recurrence*

$$T(n) = aT(n/b) + cn^k,$$

where n/b *means either* $\lfloor n/b \rfloor$ *or* $\lceil n/b \rceil$. *Then* $T(n)$ *is asymptotically bounded as follows:*

1. $T(n) = \Theta(n^{\log_b a})$ *if* $\log_b a > k$.
2. $T(n) = \Theta(n^k \log n)$ *if* $\log_b a = k$.
3. $T(n) = \Theta(n^k)$ *if* $\log_b a < k$.

Example: running time of Merge-Sort

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