Analysis of Algorithms, I
CSOR W4231

Eleni Drinea
Computer Science Department

Columbia University

Asymptotic notation, mergesort, recurrences
1. Asymptotic notation

2. The divide & conquer principle; application: mergesort

3. Solving recurrences and running time of mergesort
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: high-level language steps will expand by a constant factor that depends on the hardware.
1. Asymptotic notation

2. The divide & conquer principle; application: mergesort

3. Solving recurrences and running time of mergesort
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 input size $n$.

- 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:** Show that $T(n) = O(f(n))$ when
- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^2$.
- $T(n) = an^2 + b$ and $f(n) = n^3$. 
Definition 2 (\( \Omega \)).

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 \( T(n) \geq c \cdot f(n) \).
**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:** Show that $T(n) = \Omega(f(n))$ when

- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^2$.
- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n$. 
Asymptotic tight bounds: $\Theta$ notation

**Definition 3 ($\Theta$).**

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: $\Theta$ notation

**Definition 3 ($\Theta$).**

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: $\Theta$ notation

**Definition 3 ($\Theta$).**

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

**Notational convention:** $\log n$ stands for $\log_2 n$

**Examples:** Show that $T(n) = \Theta(f(n))$ when

- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^2$
- $T(n) = n \log n + n$ and $f(n) = n \log n$
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: Show that \( T(n) = o(f(n)) \) when

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

Definition 5 ($\omega$).

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 $\lim_{n \to \infty} \frac{T(n)}{f(n)} = \infty$, if the limit exists. Then $f(n) = o(T(n))$.

**Examples:** Show that $T(n) = \omega(f(n))$ when

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

$$\log n < n < n \log n < n^2 < 2^n < 3^n < n^n$$
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 $i$, $f_i = O(h)$.

3. Transpose symmetry
   - $f = O(g)$ if and only if $g = \Omega(f)$.
   - $f = o(g)$ if and only if $g = \omega(f)$. 
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 to get the solution to the overall 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
endif

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

mergesort \((A, left, mid)\)

mergesort \((A, mid + 1, right)\)

merge \((A, left, right, mid)\)

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?
**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 \texttt{merge}(A, left, right, mid).
Optional exercise 1: write detailed pseudocode or actual code for merge

Optional exercise 2: write a recursive merge
Analysis of merge

1. Correctness

2. Running time

3. Space
1. **Correctness:** by induction on the size of the two lists (recommended exercise)

2. **Running time**

3. **Space**
merge: pseudocode

merge \((A, left, right, mid)\)

\[ L = A[left, mid] \quad \rightarrow \text{not a primitive computational step!} \]
\[ R = A[mid + 1, right] \quad \rightarrow \text{not a primitive computational step!} \]

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.

\textbf{Remark:} the output is stored directly in \(A[left, right]\), thus the subarray \(A[left, right]\) is sorted after \(\text{merge}(A, left, right, mid)\).
Analysis of merge: running time

1. **Correctness:** by induction on the size of the two lists (recommended exercise)

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**
1. **Correctness:** by induction on the size of the two lists *(recommended exercise)*

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 $A$).
Exercise (recommended): run 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 \( n = 2^k \) for integer \( k \geq 0 \).
We will use induction on \( k \).

- **Base case:** For \( k = 0 \), the input consists of 1 item; \texttt{mergesort} returns the item.

- **Induction Hypothesis:** For \( k \geq 0 \), assume that \texttt{mergesort} correctly sorts any list of size \( 2^k \).

- **Induction Step:** We will show that \texttt{mergesort} correctly sorts any list \( A \) of size \( 2^{k+1} \).

From the pseudocode of \texttt{mergesort}, we have:

- Line 3: \( mid \) takes the value \( 2^k \)
- Line 4: \texttt{mergesort}(\( A, 1, 2^k \)) correctly sorts the leftmost half of the input, by the induction hypothesis.
- Line 5: \texttt{mergesort}(\( A, 2^k + 1, 2^{k+1} \)) correctly sorts the rightmost half of the input, by the induction hypothesis.
- Line 6: \texttt{merge} correctly merges its two sorted input lists into one sorted output of size \( 2^k + 2^k \).

\( \Rightarrow \) \texttt{mergesort} correctly sorts any input of size \( 2^{k+1} \).
Running time of \texttt{mergesort}

The running time of \texttt{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 \textit{recurrence relations}

- an \textbf{inequality} or \textbf{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 \(n\)

\textbf{Remarks}

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

1. Asymptotic notation

2. The divide & conquer principle; application: mergesort

3. Solving recurrences and running time of mergesort
The technique consists of three steps

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

**Example:** give an asymptotic bound for the recurrence describing the running time of **mergesort**

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

\[ T(1) = c \]
The running times of many recursive algorithms can be expressed by the following recurrence

\[ T(n) = aT\left(\frac{n}{b}\right) + cn^k, \text{ for } a, c > 0, b > 1, k \geq 0 \]

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

\[ \Rightarrow \text{ at level } i, \text{ there are } a^i \text{ subproblems, each of size } \frac{n}{b^i} \]

\[ \Rightarrow \text{ each subproblem at level } i \text{ requires } c\left(\frac{n}{b^i}\right)^k \text{ work} \]

- \( \text{the height of the tree is } \log_b n \text{ levels} \)

\[ \Rightarrow \text{ Total work: } \sum_{i=0}^{\log_b n} a^i c\left(\frac{n}{b^i}\right)^k = cn^k \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i \]
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

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

Example: running time of mergesort

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

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** is just a variant of simple induction where the induction step at \( n \) requires that the inductive hypothesis holds at all previous steps \( 1, 2, \ldots, n - 1 \).
How would you solve...

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

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

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