Algorithms 2020: Analysis of Algorithms

(Based on [Manber 1989])

Yih-Kuen Tsay

September 29, 2020

1 Introduction

Introduction

- The purpose of algorithm analysis is to predict the behavior (running time, space requirement, etc.) of an algorithm *without implementing it* on a specific computer. (Why?)
- As the exact behavior of an algorithm is hard to predict, the analysis is usually an *approximation*:
  - Relative to the input size (usually denoted by n): input possibilities too enormous to elaborate
  - Asymptotic: should care more about larger inputs
  - Worst-Case: easier to do, often representative (Why not average-case?)
- Such an approximation is usually good enough for comparing different algorithms for the same problem.

Complexity

- Theoretically, "complexity of an algorithm" is a more precise term for "approximate behavior of an algorithm".
- Two most important measures of complexity:
  - Time Complexity an upper bound on the number of steps that the algorithm performs.
  - Space Complexity an upper bound on the amount of temporary storage required for running the algorithm (excluding the input, the output, and the program itself).
- We will focus on time complexity.

Comparing Running Times

- How do we compare the following running times?
  1. $100n$
  2. $2n^2 + 50$
  3. $100n^{1.8}$
- We will study an approach (the O notation) that allows us to ignore constant factors and concentrate on the behavior as n goes to infinity.
- For most algorithms, the constants in the expressions of their running times tend to be small.

2 The O Notation

The O Notation

- A function $g(n)$ is $O(f(n))$ for another function $f(n)$ if there exist constants c and N such that, for all $n \geq N$, $g(n) \leq cf(n)$.
- The function $g(n)$ may be substantially less than $cf(n)$; the O notation bounds it *only from above*.
- The O notation allows us to ignore constants conveniently.
- Examples:
  - $5n^2 + 15 = O(n^2)$. (cf. $5n^2 + 15 \leq O(n^2)$ or $5n^2 + 15 \in O(n^2)$)
  - $5n^2 + 15 = O(n^3)$. (cf. $5n^2 + 15 \leq O(n^3)$ or $5n^2 + 15 \in O(n^3)$)
  - As part of an expression like $T(n) = 3n^2 + O(n)$.

The O Notation (cont.)

- No need to specify the base of a logarithm.
  - $\log_2 n = \frac{\log_{10} n}{\log_{10} 2} = \frac{1}{\log_{10} 2} \log_{10} n$.
  - For example, we can just write $O(\log n)$.
- $O(1)$ denotes a constant.

Properties of O

- We can add and multiply with O.

  Lemma 1 (3.2)**.** *1. If $f(n) = O(s(n))$ and $g(n) = O(r(n))$, then $f(n) + g(n) = O(s(n) + r(n))$. 2. If $f(n) = O(s(n))$ and $g(n) = O(r(n))$, then $f(n) \cdot g(n) = O(s(n) \cdot r(n))$.*

  /* There exist constants c_1, N_1, c_2, and N_2 such that, for all $n \geq N_1$, $f(n) \leq c_1 s(n)$ and, for all $n \geq N_2$, $g(n) \leq c_2 r(n)$. Assume without loss of generality that $c_1 \geq c_2$ and $N_1 \geq N_2$. Then, for all $n \geq N_1$, $f(n) + g(n) \leq c_1 s(n) + c_2 r(n) \leq c_1 s(n) + c_1 r(n) = c_1(s(n) + r(n))$, i.e., $f(n) + g(n) = O(s(n) + r(n))$. Also, for all $n \geq N_1$, $f(n) \cdot g(n) \leq c_1 s(n) \cdot c_2 r(n) = c_1 c_2 (s(n) \cdot r(n))$, which implies that there exist constants c and N such that, for all $n \geq N$, $f(n) \cdot g(n) \leq c(s(n) \cdot r(n))$, i.e., $f(n) \cdot g(n) = O(s(n) \cdot r(n))$. */

- However, we cannot subtract or divide with O.
  - $2n = O(n)$, $n = O(n)$, and $2n - n = n \neq O(n - n)$.
  - $n^2 = O(n^2)$, $n = O(n^2)$, and $n^2/n = n \neq O(n^2/n^2)$.

3 Speed of Growth

Polynomial vs. Exponential

- A function $f(n)$ is *monotonically growing* (or *monotonically increasing*) if $n_1 \geq n_2$ implies that $f(n_1) \geq f(n_2)$.
- An exponential function grows *at least* as fast as a polynomial function does.

Theorem 2 (3.1)**.** *For all constants $c > 0$ and $a > 1$, and for all monotonically growing functions $f(n)$, $(f(n))^c = O(a^{f(n)})$.*

- Examples:
  - Take n as $f(n)$, $n^c = O(a^n)$.
  - Take $\log_a n$ as $f(n)$, $(\log_a n)^c = O(a^{\log_a n}) = O(n)$.

Speed of Growth

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

Table: Function values.

Source: redrawn from [E. Horowitz *et al.* 1998, Table 1.7].

Speed of Growth (cont.)

running times	$time_1$ 1000 steps/sec	$time_2$ 2000 steps/sec	$time_3$ 4000 steps/sec	$time_4$ 8000 steps/sec
$\log n$	0.010	0.005	0.003	0.001
n	1	0.5	0.25	0.125
$n \log n$	10	5	2.5	1.25
$n^{1.5}$	32	16	8	4
n^2	1000	500	250	125
n^3	1,000,000	500,000	250,000	125,000
1.1^n	10^{39}	10^{39}	10^{38}	10^{38}

Table: Running times (in seconds) under different assumptions (n = 1000).

Source: redrawn from [Manber 1989, Table 3.1].

O, o, Ω, and Θ

- Let $T(n)$ be the number of steps required to solve a given problem for input size n.
- We say that $T(n) = \Omega(g(n))$ or the problem has a lower bound of $\Omega(g(n))$ if there exist constants c and N such that, for all $n \geq N$, $T(n) \geq cg(n)$.
- If a certain function $f(n)$ satisfies both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then we say that $f(n) = \Theta(g(n))$.
- We say that $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Polynomial vs. Exponential (cont.)

- An exponential function grows *faster* than a polynomial function does.

  Theorem 3 (3.3)**.** *For all constants $c > 0$ and $a > 1$, and for all monotonically growing functions $f(n)$, we have*
  $$(f(n))^c = o(a^{f(n)}).$$

- Consider a previous example again: Take $\log_a n$ as $f(n)$. For all $c > 0$ and $a > 1$,
  $$(\log_a n)^c = o(a^{\log_a n}) = o(n).$$

4 Sums

Sums

- Techniques for summing expressions are essential for complexity analysis.
- For example, given that we know

  $$S_0(n) = \sum_{i=1}^{n} 1 = n$$

  and

  $$S_1(n) = \sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2},$$

  we want to compute the sum

  $$S_2(n) = \sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2.$$

Sums (cont.)

From

$$(i+1)^3 = i^3 + 3i^2 + 3i + 1,$$

we have

$$(i+1)^3 - i^3 = 3i^2 + 3i + 1.$$

$$\begin{array}{rcl}
2^3 - 1^3 & = & 3 \times 1^2 + 3 \times 1 + 1 \\
3^3 - 2^3 & = & 3 \times 2^2 + 3 \times 2 + 1 \\
4^3 - 3^3 & = & 3 \times 3^2 + 3 \times 3 + 1 \\
\cdots & \cdots & \cdots \\
(n+1)^3 - n^3 & = & 3 \times n^2 + 3 \times n + 1 \\
\hline
(n+1)^3 - 1 & = & 3 \times S_2(n) + 3 \times S_1(n) + S_0(n) \\
(S_3(n+1) - S_3(1)) - S_3(n) & = & 3 \times S_2(n) + 3 \times S_1(n) + S_0(n)
\end{array}$$

Sums (cont.)

- So, we have

  $$(n+1)^3 - 1 = 3 \times S_2(n) + 3 \times S_1(n) + S_0(n).$$

- Given $S_0(n)$ and $S_1(n)$, the sum $S_2(n)$ can be computed by straightforward algebra.
- Recall that the left-hand side $(n+1)^3 - 1$ equals $(S_3(n+1) - S_3(1)) - S_3(n)$, a result from "shifting and canceling" terms of two sums.
- This generalizes to $S_k(n)$, for $k > 2$.
- Similar shifting and canceling techniques apply to other kinds of sums.

/* We actually will need to obtain an upper bound for the sum of n upper bounds. For instance, $\sum_{i=1}^{n} O(1) = O(\sum_{i=1}^{n} 1) = O(n)$, $\sum_{i=1}^{n} O(i) = O(\sum_{i=1}^{n} i) = O(\frac{n(n+1)}{2}) = O(n^2)$, etc. */

5 Recurrence Relations

Recurrence Relations

- A *recurrence relation* is a way to define a function by an expression involving the same function.
- The Fibonacci numbers, for example, can be defined as follows:

$$\begin{cases} F(1) = 1 \\ F(2) = 1 \\ F(n) = F(n-2) + F(n-1) \end{cases}$$

  We would need $k-2$ steps to compute $F(k)$.
- It is more convenient to have an explicit (or closed-form) expression.
- To obtain the explicit expression is called *solving* the recurrence relation.

Guessing and Proving an Upper Bound

- Recurrence relation: $\begin{cases} T(2) = 1 \\ T(2n) \leq 2T(n) + 2n - 1 \end{cases}$
- Guess: $T(n) = O(n \log n)$.
- Proof:
  1. Base case: $T(2) \leq 2 \log 2$.
  2. Inductive step: $T(2n) \leq 2T(n) + 2n - 1$

$$\begin{aligned} &\leq 2(n \log n) + 2n - 1 \\ &= 2n \log n + 2n \log 2 - 1 \\ &\leq 2n(\log n + \log 2) \\ &= 2n \log 2n \end{aligned}$$

Solving the Fibonacci Relation

- We will study two techniques for solving the Fibonacci relation.
  1. One uses the characteristic equation
  2. The other uses generating functions
- These techniques may be generalized to handle recurrence relations of the form

$$F(n) = b_1 F(n-1) + b_2 F(n-2) + \cdots + b_k F(n-k)$$

  for a constant k.

Using the Characteristic Equation

- $F(n)$ nearly doubles every time and should be an exponential function.
- But what is the base of the exponential function?
- The base a should satisfy $a^n = a^{n-1} + a^{n-2}$, which implies $a^2 = a + 1$ (called the characteristic equation).
- There are two solutions to the characteristic equation: $a_1 = \frac{1+\sqrt{5}}{2}$ and $a_2 = \frac{1-\sqrt{5}}{2}$.
- Any linear combination of a_1^n and a_2^n solves the recurrence relation.

Using the Characteristic Equation (cont.)

- So, the general solution is
$$c_1(\frac{1+\sqrt{5}}{2})^n + c_2(\frac{1-\sqrt{5}}{2})^n.$$
- To fit the values of $F(1)$ and $F(2)$, c_1 and c_2 must satisfy
$$c_1(\tfrac{1+\sqrt{5}}{2}) + c_2(\tfrac{1-\sqrt{5}}{2}) = 1$$
$$c_1(\tfrac{1+\sqrt{5}}{2})^2 + c_2(\tfrac{1-\sqrt{5}}{2})^2 = 1$$
- Therefore, $c_1 = \frac{1}{\sqrt{5}}$ and $c_2 = -\frac{1}{\sqrt{5}}$, and the exact solution to the Fibonacci relation is
$$F(n) = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n.$$

Using Generating Functions

- *Generating functions* provide a systematic, effective means for representing and manipulating infinite sequences (of numbers).
- We use them here to derive a closed-form representation of the Fibonacci numbers.
- Below are two basic generating functions:

gen. func.	power series	generated sequence
$\frac{1}{1-z}$	$1 + z + z^2 + \cdots + z^n + \cdots$	$1, 1, 1, \cdots, 1, \cdots$
$\frac{c}{1-az}$	$c + caz + ca^2z^2 + \cdots + ca^nz^n + \cdots$	$c, ca, ca^2, \cdots, ca^n, \cdots$

- The second one is a generalization of the first and will be used in our solution.

Using Generating Functions (cont.)

Let $F(z) = 0 + F_1z + F_2z^2 + F_3z^3 + \cdots + F_nz^n + \cdots$ (a generating function for the Fibonacci numbers; $F(n)$ is written as F_n here).

$$\begin{array}{rcl} F(z) & = & F_1z + F_2z^2 + F_3z^3 + \cdots + F_nz^n + F_{n+1}z^{n+1} + \cdots \\ zF(z) & = & F_1z^2 + F_2z^3 + \cdots + F_{n-1}z^n + F_nz^{n+1} + \cdots \\ z^2F(z) & = & F_1z^3 + F_2z^4 + \cdots + F_{n-2}z^n + F_{n-1}z^{n+1} + \cdots \\ \hline (1-z-z^2)F(z) & = & z \end{array}$$

$$\begin{aligned} F(z) &= \tfrac{z}{1-z-z^2} \quad (= \tfrac{z}{(1-\frac{1+\sqrt{5}}{2}z)(1-\frac{1-\sqrt{5}}{2}z)}) \\ &= \tfrac{\frac{1}{\sqrt{5}}}{1-\frac{1+\sqrt{5}}{2}z} + \tfrac{-\frac{1}{\sqrt{5}}}{1-\frac{1-\sqrt{5}}{2}z} \end{aligned}$$

/*

$$\begin{aligned} F(z) &= \tfrac{\frac{1}{\sqrt{5}}}{1-\frac{1+\sqrt{5}}{2}z} + \tfrac{-\frac{1}{\sqrt{5}}}{1-\frac{1-\sqrt{5}}{2}z} \\ &= (\tfrac{1}{\sqrt{5}} + \tfrac{1}{\sqrt{5}}\tfrac{1+\sqrt{5}}{2}z + \tfrac{1}{\sqrt{5}}(\tfrac{1+\sqrt{5}}{2})^2z^2 + \cdots + \tfrac{1}{\sqrt{5}}(\tfrac{1+\sqrt{5}}{2})^nz^n + \cdots) + \\ &\quad (-\tfrac{1}{\sqrt{5}} + (-\tfrac{1}{\sqrt{5}})\tfrac{1-\sqrt{5}}{2}z + (-\tfrac{1}{\sqrt{5}})(\tfrac{1-\sqrt{5}}{2})^2z^2 + \cdots + (-\tfrac{1}{\sqrt{5}})(\tfrac{1-\sqrt{5}}{2})^nz^n + \cdots) \\ &= z + z^2 + \cdots + (\tfrac{1}{\sqrt{5}}(\tfrac{1+\sqrt{5}}{2})^n - \tfrac{1}{\sqrt{5}}(\tfrac{1-\sqrt{5}}{2})^n)z^n + \cdots \end{aligned}$$

*/

Therefore, $F_n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n$.

6 Divide and Conquer Relations

Divide and Conquer Relations

- The running time $T(n)$ of a divide-and-conquer algorithm satisfies

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

  where

  - a is the number of subproblems,
  - n/b is the size of each subproblem, and
  - $O(n^k)$ is the time spent on dividing the problem and combining the solutions.

Divide and Conquer Relations (cont.)

Assume, for simplicity, $n = b^m$ ($\frac{n}{b^m} = 1$, $\frac{n}{b^{m-1}} = b$, etc.).

$$\begin{aligned}
T(n) \quad &= aT(\tfrac{n}{b}) + O(n^k) \\
&= a(aT(\tfrac{n}{b^2}) + O((\tfrac{n}{b})^k)) + O(n^k) \\
&= a(a(aT(\tfrac{n}{b^3}) + O((\tfrac{n}{b^2})^k)) + O((\tfrac{n}{b})^k)) + O(n^k) \\
&\cdots \\
&= a(a(\cdots(aT(\tfrac{n}{b^m}) + O((\tfrac{n}{b^{m-1}})^k)) + \cdots) + O((\tfrac{n}{b})^k)) + O(n^k)
\end{aligned}$$

Assuming $T(1) = O(1)$ (and recalling $n = b^m$, i.e., $m = \log_b n$),

$$T(n) = a^m \times O(1) + \sum_{i=1}^{m} a^{m-i} O(b^{ik}) = O(a^m) + a^m \sum_{i=1}^{m} O((\frac{b^k}{a})^i).$$

Divide and Conquer Relations (cont.)

As $m = \log_b n$ and $a^m = a^{\log_b n} = n^{\log_b a}$,

$$T(n) = O(n^{\log_b a}) + O(n^{\log_b a}) \times O(\sum_{i=1}^{\log_b n} (\frac{b^k}{a})^i).$$

- $O(n^{\log_b a})$ is the accumulative time for computing all the subproblems.
- $O(n^{\log_b a}) \times O(\sum_{i=1}^{\log_b n} (\frac{b^k}{a})^i)$ is the accumulative time for dividing problems and combining solutions.
- Three cases to consider: $\frac{b^k}{a} < 1$, $\frac{b^k}{a} = 1$, and $\frac{b^k}{a} > 1$.

/* Case 1: $\frac{b^k}{a} < 1$. The geometric series $\sum_{i=1}^{\log_b n} (\frac{b^k}{a})^i$ converges to some constant. So, $T(n) = O(n^{\log_b a}) + O(n^{\log_b a}) \times O(1) = O(n^{\log_b a})$.

Case 2: $\frac{b^k}{a} = 1$, i.e., $\log_b a = k$. $O(\sum_{i=1}^{\log_b n} (\frac{b^k}{a})^i) = O(\log_b n) = O(\log n)$. So, $T(n) = O(n^{\log_b a}) + O(n^{\log_b a}) \times O(\log n) = O(n^k \log n)$.

Case 3: $\frac{b^k}{a} > 1$. $O(\sum_{i=1}^{\log_b n} (\frac{b^k}{a})^i) = O(\frac{b^k}{a} \frac{(\frac{b^k}{a})^{\log_b n} - 1}{\frac{b^k}{a} - 1}) = O((\frac{b^k}{a})^{\log_b n}) = O(\frac{(b^k)^{\log_b n}}{a^{\log_b n}}) = O(\frac{(b^{\log_b n})^k}{n^{\log_b a}}) = O(\frac{n^k}{n^{\log_b a}})$. $T(n) = O(n^{\log_b a}) + O(n^{\log_b a}) \times O(\frac{n^k}{n^{\log_b a}}) = O(n^{\log_b a}) + O(n^k) = O(n^k)$, since $\frac{b^k}{a} > 1$ implies $k > \log_b a$. */

Divide and Conquer Relations (cont.)

Theorem 4 (3.4)**.** *The solution of the recurrence relation $T(n) = aT(n/b) + O(n^k)$, where a and b are integer constants, $a \geq 1$, $b \geq 2$, and k is a non-negative real constant, is*

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

This theorem may be slightly generalized by replacing $O(n^k)$ with some $f(n)$, but the current form is sufficient for the cases we will encounter. Due to its generality and usefulness, the theorem has conventionally been referred to as "the master theorem".

/* Example 1: Suppose $T(n) = T(n/2) + O(1)$ (arising from, e.g., binary search). In this case, $a = 1$, $b = 2$, and $k = 0$. We have $a = b^k$ and the second case of the theorem applies. Therefore, $T(n) = O(n^0 \log n) = O(\log n)$.

Example 2: Suppose $T(n) = 2T(n/2) + O(n)$ (arising from, e.g., merge sort). In this case, $a = 2$, $b = 2$, and $k = 1$. We have $a = b^k$ and again the second case of the theorem applies. Therefore, $T(n) = O(n \log n)$. */

Recurrent Relations with Full History

- Example One:

$$T(n) = c + \sum_{i=1}^{n-1} T(i),$$

where c is a constant and $T(1)$ is given separately.

- $T(n) - T(n-1) = (c + \sum_{i=1}^{n-1} T(i)) - (c + \sum_{i=1}^{n-2} T(i)) = T(n-1)$; hence, $T(n) = 2T(n-1)$. (This holds only for $n \geq 3$.) /* The relation $T(n) = 2T(n-1)$ does not hold for $n = 2$, as $T(2) - T(1) = c$ (not $T(1)$). */
- So, we get

$$\begin{cases} T(2) = c + T(1) \\ T(n) = 2T(n-1) & \text{if } n \geq 3 \end{cases}$$

which is easier to solve.

- $T(n+1) = (c + T(1))2^{n-1}$, for $n \geq 2$.

Recurrent Relations with Full History (cont.)

- Example Two:

$$T(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} T(i), (\text{for} n \geq 2). T(1) = 0.$$

- Multiply both sides of the equation with n for $T(n)$ and $(n+1)$ for $T(n+1)$.

$$nT(n) = n(n-1) + 2\sum_{i=1}^{n-1} T(i)$$
$$(n+1)T(n+1) = (n+1)n + 2\sum_{i=1}^{n} T(i)$$

- Take the difference.

$$(n+1)T(n+1) - nT(n) = (n+1)n - n(n-1) + 2T(n) = 2n + 2T(n)$$

which implies

$$T(n+1) = \frac{n+2}{n+1}T(n) + \frac{2n}{n+1}$$

Recurrent Relations with Full History (cont.)

- Further simplification.

$$T(n+1) \leq \frac{n+2}{n+1}T(n) + 2$$

- Expanding and canceling.

$$\begin{aligned}
&T(n)\\
&\leq 2 + \tfrac{n+1}{n}(2 + \tfrac{n}{n-1}(2 + \tfrac{n-1}{n-2}(\cdots(2 + \tfrac{4}{3}T(2))\cdots)))\\
&\leq 2(1 + \tfrac{n+1}{n} + \tfrac{n+1}{n}\tfrac{n}{n-1} + \tfrac{n+1}{n}\tfrac{n}{n-1}\tfrac{n-1}{n-2} + \cdots + (\tfrac{n+1}{n}\tfrac{n}{n-1}\cdots\tfrac{4}{3}))\\
&\leq 2(n+1)(\tfrac{1}{n+1} + \tfrac{1}{n} + \tfrac{1}{n-1} + \cdots + \tfrac{1}{3})\\
&\leq 2 + 2(n+1)(\tfrac{1}{n} + \tfrac{1}{n-1} + \cdots + 1)\\
&= O(n\log n)
\end{aligned}$$

(Note: $T(1) = 0$ and $T(2) \leq 2 + \frac{3}{2}T(1) = 2$)

7 Useful Facts

Useful Facts

- Bounding a summation by an integral:
  If $f(x)$ is monotonically *increasing*, then

$$\sum_{i=1}^{n} f(i) \leq \int_{1}^{n+1} f(x)dx.$$

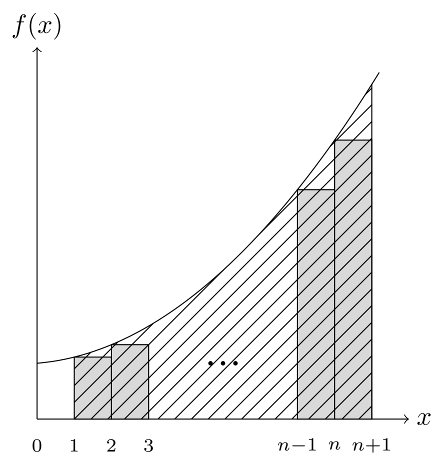
  If $f(x)$ is monotonically *decreasing*, then

$$\sum_{i=1}^{n} f(i) \leq f(1) + \int_{1}^{n} f(x)dx.$$

- Stirling's approximation

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n(1 + O(1/n)).$$

Bounding a Summation by an Integral



$$\sum_{i=1}^{n} f(i) \leq \int_{1}^{n+1} f(x)dx.$$

/* This technique can be used to show that $\int_0^n f(x)dx \leq \sum_{i=1}^n f(i)$, by shifting the n vertical bars (which represent $\sum_{i=1}^n f(i)$) in the diagram to the left by one unit.

When $f(x)$ is monotonically decreasing, we state that $\sum_{i=1}^n f(i) \leq f(1) + \int_1^n f(x)dx$, rather than $\sum_{i=1}^n f(i) \leq \int_0^n f(x)dx$, as the part $\int_0^1 f(x)dx$ might go to infinity and would not be a good upper bound. Isolating the first term of the sum, we have $\sum_{i=1}^n f(i) = f(1) + \sum_{i=2}^n f(i) \leq f(1) + \int_1^n f(x)dx$. It can also be shown that $\int_1^{n+1} f(x)dx \leq \sum_{i=1}^n f(i)$. */

Useful Facts (cont.)

- Harmonic series
$$H_n = \sum_{k=1}^{n} \frac{1}{k} = \ln n + \gamma + O(1/n),$$
where $\gamma = 0.577\ldots$ is Euler's constant. So, $H_n = O(\log n)$.

  /* The upper bound may also be obtained using an integral. $\sum_{k=1}^n \frac{1}{k} \leq \frac{1}{1} + \int_1^n \frac{1}{x}dx = 1 + \ln n = O(\ln n) = O(\log n)$. */

- Sum of logarithms
$$\begin{aligned} \textstyle\sum_{i=1}^n \lfloor \log_2 i \rfloor &= (n+1)\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2 \\ &= \Theta(n \log n). \end{aligned}$$

  /* $\sum_{i=1}^n \lfloor \log_2 i \rfloor \leq \sum_{i=1}^n \log_2 i = \log_2(n!) = \log_2(\sqrt{2\pi n}(\frac{n}{e})^n(1 + O(1/n))) = O(\log_2(\sqrt{2\pi n}(\frac{n}{e})^n)) = O(\log_2 \sqrt{2\pi n} + \log_2(\frac{n}{e})^n) = O(\log_2 \sqrt{2\pi n} + n \log_2(\frac{n}{e})) = O(n \log n)$. The other direction $\sum_{i=1}^n \lfloor \log_2 i \rfloor \geq (\sum_{i=1}^n \log_2 i) - n$. */