COMP3711

Design and Analysis of Algorithms

Larry128

A summary notes for revision

Fall 2024-2025

Contents

1 Prerequisites

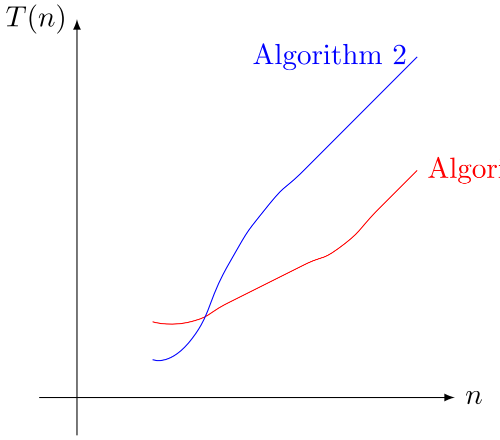
1.1 Input size of Problems

Input size	how large the input is.
Assumption	1. any number can be stored in a computer word
	2. each arithmetic operation takes constant time
Examples	Sorting: Size of the list or array
	Graph problems: Numbers of vertices and edges
	Searching: Number of input keys

1.2 Asymptotic Notation

1. Running time/ Cost of algorithms
   i. a function of input size: $T(n)$
   ii. number of operations (e.g., comparisons between two numbers)
   iii. using **asymptotic notation**, which ignores constants and non-dominant growth terms

2. Intuitions

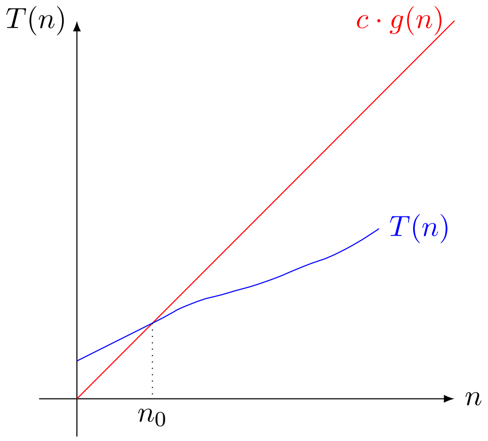


   From the figure above, Algorithm 1 is better for large n.

3. Rigorous definition of asymptotic notation

Upper bound $T(n) = O(f(n))$	if $\exists c > 0$ and $n_0 \geq 0$ such that $\forall n \neq n_0, T(n) \leq cf(n)$
Lower bound $T(n) = \Omega(f(n))$	if $\exists c > 0$ and $n_0 \neq 0$ such that $\forall n \neq n_0, T(n) \geq cf(n)$
Tight bound $T(n) = \Theta(f(n))$	if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

4. Big-O Notation

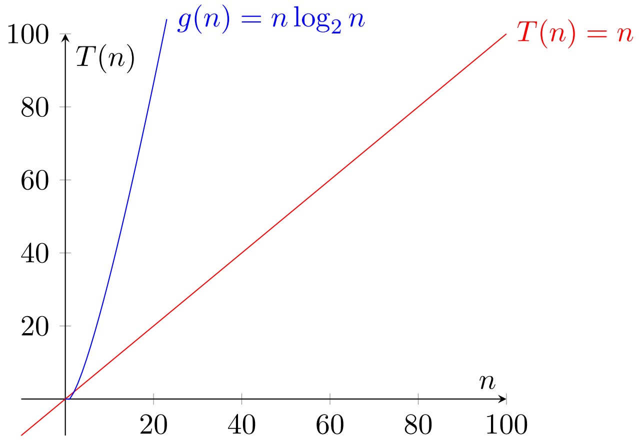


$$T(n) = O(g(n)) \iff \exists n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq T(n) \leq c \cdot g(n)$$

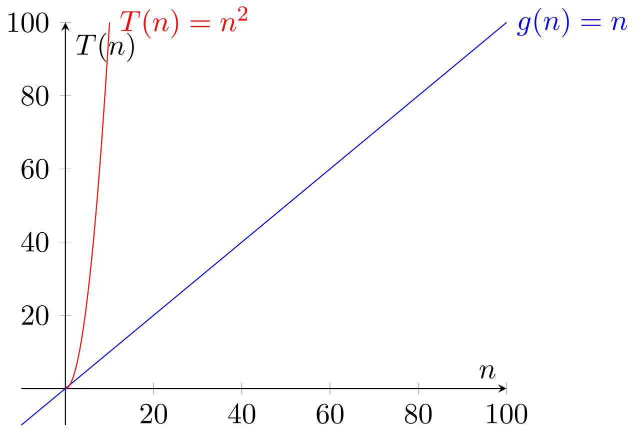
   Below are some examples of Big-O notation proofs

(a) $T(n) = n, g(n) = n\log_2 n$
We wish to proof $T(n) = n \in O(n\log_2 n)$.
Choose $c = 1, n_0 = 2$, for all $n \geq 2 = n_0$,

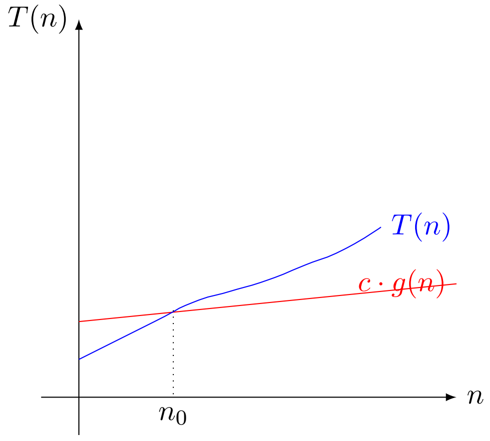
$$1 \leq \log_2 n \iff n \leq n\log_2 n \iff n \leq c \cdot n\log_2 n$$



(b) $T(n) = n^2, g(n) = n$
We wish to proof $T(n) = n^2 \notin O(g(n))$ by contradiction.
Suppose there exists some c and n_0 such that for all $n \geq n_0$, $n^2 \leq c \cdot n$. Then, $n \leq c$, $\forall n \geq n_0$, which is not possible as c is a constant and n can be arbitrarily large.

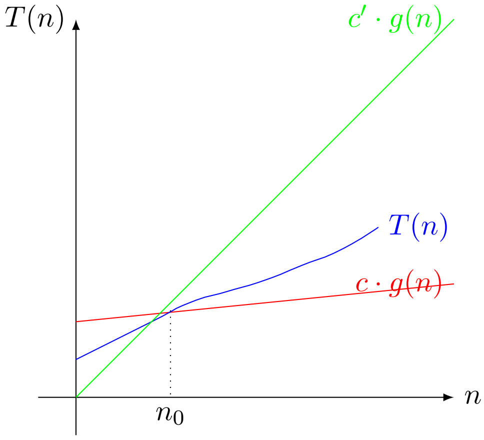


5. Big-Ω Notation



$$T(n) = \Omega(g(n)) \iff \exists n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq c \cdot g(n) \leq T(n)$$

6. Big-Θ Notation



$$T(n) = \Theta(f(n)) \iff T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n))$$

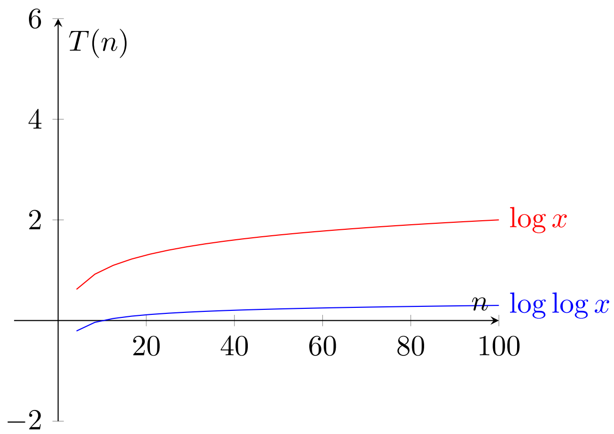
7. Implementation and experimentation are needed sometimes
   If algorithm A is $T_1(n) = 10n \in \Theta(n)$, algorithm B is $T_2(n) = 1000n \in \Theta(n)$, but algorithm A is superior in practice. In this case, Implementation and experimentation are needed.

8. Basic facts on exponents and logarithms
   (a) $2^{2n} \neq \Theta(2^n)$, proof: set $x = 2^n$, then $x^2 \neq \Theta(x)$
   (b) $2^{n+2} = 4 \cdot 2^n = \Theta(2^n)$
   (c) $\log_a(n^b) = \dfrac{b \log n}{\log a} = \Theta(\log n)$
   (d) $\log_b a = \dfrac{1}{\log_a b}$
   (e) $a^{\log_b n} = n^{\log_b a}$

9. Important note on growth of functions

$$k < \log n < n^a < n \log n < n^b < c^n$$

   ,where $k, c \in \mathbb{R}, 0 < a < 2, b \geq 2$ are constants
   (a) $999^{999^{999}} = \Theta(1)$
   (b) $\log \log n = O(\log n)$, proof: for $n \geq 2$, $\log \log n \leq \log n$



   (c) $n \log n = O(\dfrac{n^2}{\log n})$

   proof: To show $n \log n = O(\dfrac{n^2}{\log n})$, it suffices to show that there exists a $C > 0$, such that

$n \log n < C \cdot \dfrac{n^2}{\log n}$ for sufficiently large n.

$$\begin{aligned} n \log n &< C \cdot \frac{n^2}{\log n} \\ \iff (\log n)^2 &< C \cdot n \end{aligned}$$

It's obvious that for large n, $\log(n) < n^\epsilon$ for $\epsilon > 0$, then we can pick $\epsilon = \dfrac{1}{2}$

$$\begin{aligned} \log n &< n^{\frac{1}{2}} \\ (\log n)^2 &< n \end{aligned}$$

Since $C > 0$, we can see $(\log n)^2 < n < C \cdot n$. We are done.

10. Extra Examples

(a) $1000n + n \log n = O(n \log n)$

(b) $n^2 + n \log(n^3) = n^2 + 3n \log n = O(n^2)$

(c) $n^3 = \Omega(n)$

(d) $n^3 = O(n^{10})$

(e) Let $f(n)$ and $g(n)$ be non-negative functions. Using basic definition of Θ-notation, proof that $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$

i. Step 1: proof $\max\{f(n), g(n)\} = O(f(n) + g(n))$
For all n, $\max\{f(n), g(n)\}$ is either equal to $f(n)$ or equal to $g(n)$. So we can deduce that $\max\{f(n), g(n)\} \leq f(n) + g(n)$. Therefore, $\max\{f(n), g(n)\} = O(f(n) + g(n))$.

ii. Step 2: proof $\max\{f(n), g(n)\} = \Omega(f(n) + g(n))$
Note that $\max\{f(n), g(n)\} \geq f(n)$ and $\max\{f(n), g(n)\} \geq g(n)$. So

$$\begin{aligned} \max\{f(n), g(n)\} + \max\{f(n), g(n)\} &\geq f(n) + g(n) \\ 2 \cdot \max\{f(n), g(n)\} &\geq f(n) + g(n) \\ \max\{f(n), g(n)\} &\geq \frac{1}{2}(f(n) + g(n)) \end{aligned}$$

Then, we have $\max\{f(n), g(n)\} = \Omega(f(n) + g(n))$

(f) if $A = \log \sqrt{n}$, $B = \sqrt{\log n}$, then $A = \Omega(B)$
proof: $A = \log \sqrt{n} = \dfrac{1}{2} \log n = \Theta(\log n)$, $B = \sqrt{\log n} = \Theta(\log n)$. We can simply deduce that $\log \sqrt{n} = \Omega(\sqrt{\log n})$

(g) Bounds of series - Arithmetic Series
Proof that $\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n = \Theta(n^2)$

i. Approach 1: use formula $\sum_{i=1}^{n} i = \dfrac{n(1+n)}{2} = \Theta(n^2)$

ii. Approach 2

A. Step 1: proof $\sum_{i=1}^{n} i = O(n^2)$

$$\begin{aligned} \sum_{i=1}^{n} i &= 1 + 2 + 3 + \cdots + (n-1) + n \\ &\leq n + n + ... + n \\ &= \sum_{i=1}^{n} n \\ &= n \cdot n \\ &= n^2 = O(n^n) \end{aligned}$$

B. Step 2: proof $\sum_{i=1}^{n} i = \Omega(n^2)$

$$\begin{aligned}\sum_{i=1}^{n} i &= 1 + 2 + 3 + \cdots + (n-1) + n \\ &\geq 0 + 0 + \cdots + 0 + ... + \frac{n}{2} + (\frac{n}{2} + 1) + \cdots + n \\ &\geq \frac{n}{2} \cdot \frac{n}{2} \\ &= \frac{n^2}{4} = \Omega(n^2)\end{aligned}$$

Then, we can say that $\sum_{i=1}^{n} i = \Theta(n^2)$

(h) Bounds of series - Polynomial Series
Proof that $\sum_{i=1}^{n} i^c = 1^c + 2^c + 3^c + \cdots + (n-1)^c + n^c = \Theta(n^{c+1})$
(The proof is more or less the same as the approach 2 of arithmetic series.)

(i) Bounds of series - Harmonic Series H_n
Proof that $H_n = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$.
Let $k = \log_2 n$, then $n = 2^k$.

index	lower bound	parts of H_n	upper bound
0	$\frac{1}{2}$	1	1
1	$2 \times \frac{1}{4} = \frac{1}{2}$	$\frac{1}{2} + \frac{1}{3}$	$2 \times \frac{1}{2} = 1$
2	$4 \times \frac{1}{8} = \frac{1}{2}$	$\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}$	$4 \times \frac{1}{4} = 1$
	$\cdots$	$\cdots$	$\cdots$
k-1	$2^{k-1} \times \frac{1}{2^k} = \frac{1}{2}$	$\frac{1}{2^{k-1}} + \frac{1}{2^{k-1}+1} \cdots + \frac{1}{2^k - 1}$	$2^{k-1} \times \frac{1}{2^{k-1}} = 1$
k	0	$\frac{1}{2^k} = \frac{1}{n}$	1

Therefore, $H_n < \sum_{i=0}^{k} 1 = k + 1 = \log_2 n + 1 = O(\log n)$ and $H_n > \sum_{i=0}^{k-1} \frac{1}{2} + 0 = \frac{k}{2} = \frac{\log_2 n}{2} = \Omega(\log n)$. So, $H_n = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$.

1.3 Introduction to Algorithms

1. What is an algorithm?
   An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions.

2. Examples of algorithms

   (a) Adding two numbers
   Input: 2 numbers $x = \overline{x_n x_{n-1} \cdots x_1}, y = \overline{y_n y_{n-1} \cdots y_1}$.
   Output: A number $z = \overline{z_{n+1} z_n \cdots z_1}$, such that $z = x + y$.

```
/*We assume x, y are arrays of length n, z is of length n+1 */
int c = 0; // offset
for (int i = 0; i < n; ++i){
   z[i] = x[i] + y[i] + c;
   if (z[i] >= 10) {
      c = 1;
      z[i] = z[i] - 10;
   }else c = 0;
}
z[n] = c;
```

   (b) Sorting Problem
   Input: An array $A[1 \cdots n]$ of elements, e.g., $[4, 8, 2, 7, 5, 6, 9, 3]$
   Output: An array $A[1 \cdots n]$ of elements in sorted order (ascending), e.g., $[2, 3, 4, 5, 6, 7, 8, 9]$

i. Selection sort

```
/* Selection sort for ascending order */
for (int i=0; i<n-1; ++i){
    // in the i-th pass, find the smallest element in A[i, i+2, ..., n]
     and swap it with A[i]
    for (int j=i+1; j<n; ++j){
        if (A[i] > A[j]){ // swap A[i] and A[j] if A[i] > A[j]
            int temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }
}
```

For example:

i = 0	(5, 2, 8, 6, 7, 1) →(2, 5, 8, 6, 7, 1) → (1, 5, 8, 6, 7, 2)
i = 1	(1, 5, 8, 6, 7, 2) → (1, 2, 8, 6, 7, 5)
i = 2	(1, 2, 8, 6, 7, 5) → (1, 2, 6, 8, 7, 5) → (1, 2, 5, 8, 7, 6)
i = 3	(1, 2, 5, 8, 7, 6) → (1, 2, 5, 7, 8, 6) → (1, 2, 5, 6, 8, 7)
i = 4	(1, 2, 5, 6, 8, 7)

Running time of selection sort

For selection sort, the total cost of algorithm (total number of comparisons) can be given by

$$(n-1) + (n-2) + \cdots + 2 + 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)(1+n-1)}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$$

Alternatively, we could think in this way: note that the algorithm runs through all possible (i, j) pairs with $1 \leq i \leq j \leq n$. There are $\binom{n}{2} = \dfrac{n(n-1)}{2}$ possible pairs. So, that's the cost of selection sort.

Note: The cost is *always* the same for any array of size n.

Proof of correctness of selection sort

Claim: When selection sort terminates, the array is sorted.

Proof: By induction on n.

When $n = 1$, the algorithm is obviously correct because there's only one element in the array.

Assume that the algorithm sorts every array of size $n-1$ correctly.

Now, consider what the algorithm does on $A[1 \cdots n]$.

A. It first puts the smallest item in $A[1]$.

B. It then runs the selection sort on $A[2 \cdots n]$ of size n. By inductive assumption, this sorts the items in $A \cdots n$.

C. Since $A[1]$ is smaller than every item in $A[2 \cdots n]$, all the items in $A[1 \cdots n]$ are now sorted.

ii. Insertion sort

```
/* Insertion sort for ascending order */
for (int i=1; i<n; ++i){
    int j= i-1;
    while (j>=0 && A[j]>A[j+1]){
        int temp = A[j];
        A[j] = A[j+1];
        A[j+1] = temp;
        j= j -1;
    }
}
```

For example:

i = 1	(5 1 8 6 3 2)→(1 5 8 6 3 2)→(1 5 8 6 3 2)
i = 2	(1 5 8 6 3 2)→(1 5 8 6 3 2)
i = 3	(1 5 8 6 3 2)→(1 5 6 8 3 2)→(1 5 6 8 3 2)
i = 4	(1 5 6 8 3 2)→(1 5 6 3 8 2)→(1 5 3 6 8 2)→(1 3 5 6 8 2)→(1 3 5 6 8 2)
i = 5	(1 3 5 6 8 2)→(1 3 5 6 2 8)→(1 3 5 2 6 8)→(1 3 2 5 6 8)→(1 2 3 5 6 8) →(1 2 3 5 6 8)

Running time of insertion sort

Total cost of insertion sort/ number of comparison is *at most*

$$\sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}$$

. This worst case happens when the input array in descending order.

Note: unlike selection sort which always uses $\frac{n(n-1)}{2}$ comparisons for each array of size n, the number of comparisons (running time) of Insertion Sort depends on the input array, and ranges between $n-1$ and $\frac{n(n-1)}{2}$.

Proof of correctness of insertion sort

$A[1 \cdots i-1]$-sorted	key	$A[i+1 \cdots n]$-unsorted

After step i, items in $A[1 \cdots i]$ are in proper order. The i-th iteration puts key $A[i]$ in proper place.

iii. Wild-Guess sort

First, we create an array with random permutation, $\vec{\pi} = [4, 7, 1, 3, 8, ...]$, of length n.

```
/* check if the order is correct or not */
bool check(const int A[], const int& n){
   for (int i=0; i<n-1; ++i){
      if ( A[pi[i]] > A[pi[i+1]] ) return false;
   }
   return true;
}
```

```
if (check(A, n)) return;
else insertion_sort(A, n);
```

It has a very small probability that wild-guess sort is faster than insertion sort but most likely it's slower.