Introduction to Algorithm*

Xiaofeng Gao

Department of Computer Science and Engineering
Shanghai Jiao Tong University, P.R.China

X033533-Algorithm: Analysis and Theory

*Special thanks is given to Prof. Yuxi Fu for sharing his teaching materials.

Outline

Outline

Algorithm

An algorithm is a procedure that consists of a finite set of *instructions* which, given an *input* from some set of possible inputs, enables us to obtain an *output* through a systematic execution of the instructions that *terminates* in a finite number of steps.

Algorithm

An algorithm is a procedure that consists of a finite set of *instructions* which, given an *input* from some set of possible inputs, enables us to obtain an *output* through a systematic execution of the instructions that *terminates* in a finite number of steps.

Theorem proving is in general not algorithmic.

Theorem verification is often algorithmic.

Quotation from Donald E. Knuth

"Computer Science is the study of algorithms."

——Donald E. Knuth

What is Computer Science?

Computer Science is the study of problem solving using computing machines. The computing machines must be physically feasible.

Donald E. Knuth
(1938 –)
Stanford University

Remark on Algorithm

The word ‘algorithm’ is derived from the name of **Muhamma ibn Mūsā al-Khwārizmī** (780?-850?), a Muslim mathematician whose works introduced Arabic numerals and algebraic concepts to Western mathematics.

The word ‘algebra’ stems from the title of his book Kitab al jahr wa’l-muqābala".

(American Heritage Dictionary)

Algorithm vs. Program

A *program* is an implementation of an algorithm, or algorithms.

A program does not necessarily terminate.

Outline

What is Computer Science?

I. **Theory of Computation** is to understand the notion of computation in a formal framework.

- Some well known models are: the general recursive function model of Gödel and Church, Church's λ-calculus, Post system model, Turing machine model, RAM, etc.

What is Computer Science?

I. **Theory of Computation** is to understand the notion of computation in a formal framework.

- Some well known models are: the general recursive function model of Gödel and Church, Church's λ-calculus, Post system model, Turing machine model, RAM, etc.

II. **Computability Theory** studies what problems can be solved by computers.

What is Computer Science?

I. **Theory of Computation** is to understand the notion of computation in a formal framework.

- Some well known models are: the general recursive function model of Gödel and Church, Church's λ-calculus, Post system model, Turing machine model, RAM, etc.

II. **Computability Theory** studies what problems can be solved by computers.

III. **Computational Complexity** studies how much resource is necessary in order to solve a problem.

What is Computer Science?

I. **Theory of Computation** is to understand the notion of computation in a formal framework.

- Some well known models are: the general recursive function model of Gödel and Church, Church's λ-calculus, Post system model, Turing machine model, RAM, etc.

II. **Computability Theory** studies what problems can be solved by computers.

III. **Computational Complexity** studies how much resource is necessary in order to solve a problem.

IV. **Theory of Algorithm** studies how problems can be solved.

Outline

Linear Search, First Example of an Algorithm

The problem to start with: Search and Ordering.

Linear Search, First Example of an Algorithm

The problem to start with: Search and Ordering.

Algorithm 1.1 LinearSearch
Input: An array $A[1..n]$ of n elements and an element x.
Output: j if $x = A[j]$, $1 \leq j \leq n$, and 0 otherwise.

1. $j \leftarrow 1$
2. **while** $j < n$ **and** $x \neq A[j]$
3. $\quad j \leftarrow j + 1$
4. **end while**
5. **if** $x = A[j]$ **then return** j **else return** 0

Binary Search

Algorithm 1.2 BinarySearch

Input: An array $A[1..n]$ of n elements sorted in nondecreasing order and an element x.

Output: j if $x = A[j]$, $1 \leq j \leq n$, and 0 otherwise.

$low \leftarrow 1$; $high \leftarrow n$; $j \leftarrow 0$
while $low \leq high$ **and** $j = 0$
    $mid \leftarrow \lfloor (low + high)/2 \rfloor$
    if $x = A[mid]$ **then** $j \leftarrow mid$ **break**
    else if $x < A[mid]$ **then** $high \leftarrow mid - 1$
    else $low \leftarrow mid + 1$
end while
return j

Analysis of BinarySearch

Suppose $x \geq 35$. A run of BinarySearch on $A[1..14]$ (see below) is

1	4	5	7	8	9	10	12	15	22	23	27	32	35

$\downarrow$

12	15	22	23	27	32	35

$\downarrow$

27	32	35

$\downarrow$

35

Analysis of BinarySearch

Analysis of BinarySearch

The complexity of the algorithm is the number of comparison.

Analysis of BinarySearch

The complexity of the algorithm is the number of comparison.

The number of comparison is maximum if $x \geq A[n]$.

Analysis of BinarySearch

The complexity of the algorithm is the number of comparison.

The number of comparison is maximum if $x \geq A[n]$.
The number of comparisons is the same as the number of iterations.

Analysis of BinarySearch

The complexity of the algorithm is the number of comparison.

The number of comparison is maximum if $x \geq A[n]$.
The number of comparisons is the same as the number of iterations.

In the second iteration, the number of elements in $A[mid + 1..n]$ is exactly $\lfloor n/2 \rfloor$.

Analysis of BinarySearch

The complexity of the algorithm is the number of comparison.

The number of comparison is maximum if $x \geq A[n]$.
The number of comparisons is the same as the number of iterations.

In the second iteration, the number of elements in $A[mid + 1..n]$ is exactly $\lfloor n/2 \rfloor$.

In the j-th iteration, the number of elements in $A[mid + 1..n]$ is exactly $\lfloor n/2^{j-1} \rfloor$.

Analysis of BinarySearch

The complexity of the algorithm is the number of comparison.

The number of comparison is maximum if $x \geq A[n]$.
The number of comparisons is the same as the number of iterations.

In the second iteration, the number of elements in $A[mid + 1..n]$ is exactly $\lfloor n/2 \rfloor$.

In the j-th iteration, the number of elements in $A[mid + 1..n]$ is exactly $\lfloor n/2^{j-1} \rfloor$.

The maximum number of iteration is the j such that $\lfloor n/2^{j-1} \rfloor = 1$, which is equivalent to $j - 1 \leq \log n < j$.

Analysis of BinarySearch

The complexity of the algorithm is the number of comparison.

The number of comparison is maximum if $x \geq A[n]$.
The number of comparisons is the same as the number of iterations.

In the second iteration, the number of elements in $A[mid+1..n]$ is exactly $\lfloor n/2 \rfloor$.

In the j-th iteration, the number of elements in $A[mid+1..n]$ is exactly $\lfloor n/2^{j-1} \rfloor$.

The maximum number of iteration is the j such that $\lfloor n/2^{j-1} \rfloor = 1$, which is equivalent to $j - 1 \leq \log n < j$.

Hence $j = \lfloor \log n \rfloor + 1$.

Merging Two Sorted Lists

Algorithm 1.3 Merge
Input: An array $A[1..m]$ of elements and three indices p, q and r. with $1 \leq p \leq q < r \leq m$, such that both the subarray $A[p..q]$ and $A[q+1..r]$ are sorted individually in nondecreasing order.
Output: $A[p..r]$ contains the result of merging the two subarrays $A[p..q]$ and $A[q+1..r]$.
Comment: $B[p..r]$ is an auxiliary array

Merging Two Sorted Lists

1. $s \leftarrow p$; $t \leftarrow q+1$; $k \leftarrow p$
2. **while** $s \leq q$ **and** $t \leq r$
3.   **if** $A[s] \leq A[t]$ **then**
4. $B[k] \leftarrow A[s]$
5. $s \leftarrow s+1$
6.   **else**
7. $B[k] \leftarrow A[t]$
8. $t \leftarrow t+1$
9.   **end if**
10. $k \leftarrow k+1$
11. **end while**
12. **if** $s = q+1$ **then** $B[k..r] \leftarrow A[t..r]$
13. **else** $B[k..r] \leftarrow A[s..q]$
13. **end if**
13. $A[p..r] \leftarrow B[p..r]$

Analysis of Merge

Suppose $A[p..q]$ has m elements and $A[q+1..r]$ has n elements. The number of comparisons done by Algorithm Merge is

Analysis of Merge

Suppose $A[p..q]$ has m elements and $A[q+1..r]$ has n elements. The number of comparisons done by Algorithm Merge is

- at least $\min\{m, n\}$;

E.g.

2	3	6

and

7	11	13	45	57

Analysis of Merge

Suppose $A[p..q]$ has m elements and $A[q+1..r]$ has n elements. The number of comparisons done by Algorithm Merge is

- at least $\min\{m, n\}$;

  E.g.

2	3	6

  and

7	11	13	45	57

- at most $m + n - 1$.

  E.g.

2	3	66

  and

7	11	13	45	57

Analysis of Merge

Suppose $A[p..q]$ has m elements and $A[q+1..r]$ has n elements. The number of comparisons done by Algorithm Merge is

- at least $\min\{m,n\}$;

  E.g.

2	3	6

  and

7	11	13	45	57

- at most $m+n-1$.

  E.g.

2	3	66

  and

7	11	13	45	57

If the two array sizes are $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, the number of comparisons is between $\lfloor n/2 \rfloor$ and $n-1$.

Outline

Selection Sort

Algorithm 1.4 SelectionSort
Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

1. **for** $i \leftarrow 1$ **to** $n - 1$
2. $\quad k \leftarrow i$
3. $\quad$ **for** $j \leftarrow i + 1$ **to** n
4. $\qquad$ **if** $A[j] < A[k]$ **then** $k \leftarrow j$
5. $\quad$ **end for**
6. $\quad$ **if** $k \neq i$ **then** interchange $A[i]$ and $A[k]$
7. **end for**

Analysis of SelectionSort

The number of comparisons carried out by Algorithm SelectionSort is precisely

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

Insertion Sort

Algorithm 1.5 InsertionSort
Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

1. **for** $i \leftarrow 2$ **to** n
2. $\quad x \leftarrow A[i]$
3. $\quad j \leftarrow i - 1$
4. $\quad$ **while** $j > 0$ **and** $A[j] > x$
5. $\quad\quad A[j+1] \leftarrow A[j]$
6. $\quad\quad j \leftarrow j - 1$
7. $\quad$ **end while**
8. $\quad A[j+1] \leftarrow x$
9. **end for**

Analysis of InsertionSort

The number of comparisons carried out by Algorithm InsertionSort is at least

$$n - 1$$

and at most

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

Bottom-Up Merge Sort

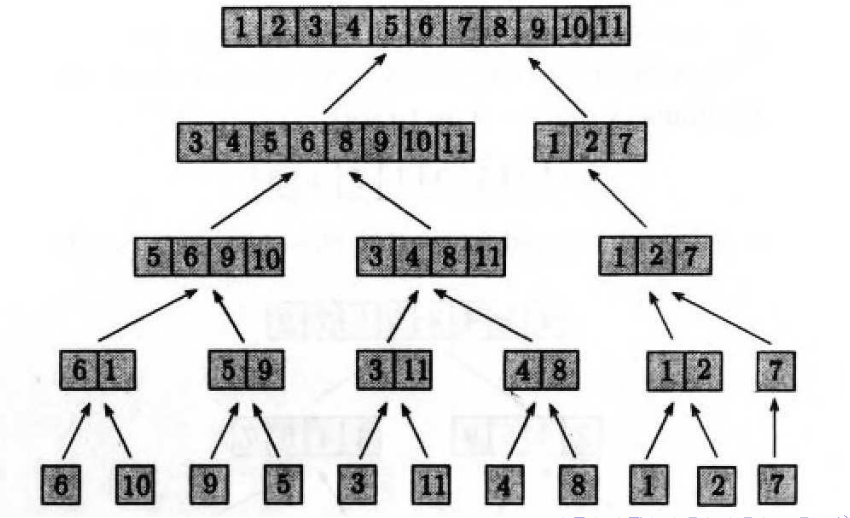
Algorithm 1.6 BottomUpSort

Input: An array $A[1..n]$ of n elements.

Output: $A[1..n]$ sorted in nondecreasing order.

```
t ← 1
while t < n
    s ← t; t ← 2s; i ← 0
    while i + t ≤ n
        Merge(A, i + 1, i + s, i + t)
        i ← i + t
    end while
    if i + s < n then Merge(A, i + 1, i + s, n)
end while
```

An Example


1 2 3 4 5 6 7 8 9 10 11
3 4 5 6 8 9 10 11
1 2 7
5 6 9 10
3 4 8 11
1 2 7
6 1
5 9
3 11
4 8
1 2
7
6
10
9
5
3
11
4
8
1
2
7

Analysis of BottomUpSort

Suppose that n is a power of 2, say $n = 2^k$.

- The outer **while** loop is executed $k = \log n$ times.
- Step 8 is never invoked.
- In the j-th iteration of the outer **while** loop, there are $2^{k-j} = n/2^j$ pairs of arrays of size 2^{j-1}.
- The number of comparisons needed in the merge of two sorted arrays in the j-th iteration is at least 2^{j-1} and at most $2^j - 1$.
- The number of comparisons in BottomUpSort is at least

$$\sum_{j=1}^{k} (\frac{n}{2^j}) 2^{j-1} = \sum_{j=1}^{k} \frac{n}{2} = \frac{n \log n}{2}$$

- The number of comparisons in BottomUpSort is at most

$$\sum_{j=1}^{k} (\frac{n}{2^j})(2^j - 1) = \sum_{j=1}^{k} (n - \frac{n}{2^j}) = n \log n - n + 1$$

Outline

Time Complexity

Computational Complexity evolved from 1960's, flourished in 1970's and 1980's.

- Time is the most precious resource.
- Important to human.

Running Time

Running time of a program is determined by:

- input size
- quality of the code
- quality of the computer system
- time complexity of the algorithm

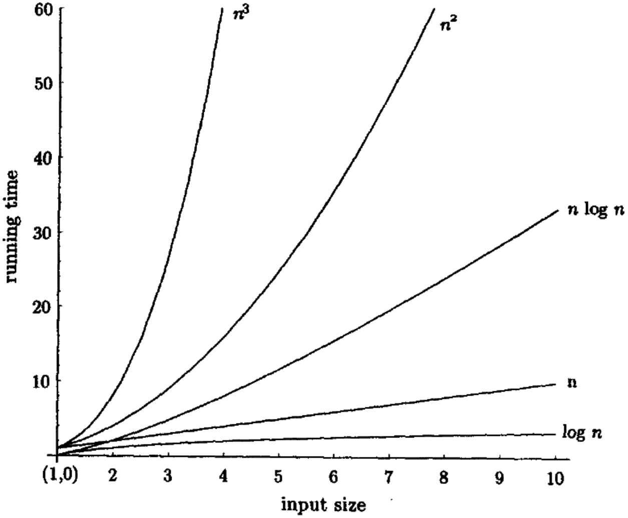
We are mostly concerned with the behavior of the algorithm under investigation on large input instances.

So we may talk about the rate of growth or the order of growth of the running time

Running Time vs Input Size

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51 μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10^{20} cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10^{58} cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10^{135} cent
2048	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10^{598} cent
4096	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10^{1214} cent
8192	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10^{2447} cent
16384	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10^{4913} cent
32768	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10^{9845} cent
65536	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	10^{19709} cent
131072	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	10^{39438} cent
262144	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	10^{78894} cent
524288	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	10^{157808} cent
1048576	0.02 μ	1048.60 μ	20972 μ	18.3 min	37 years	10^{315634} cent

Growth of Typical Functions



Elementary Operation

Definition: We denote by an “elementary operation" any computational step whose cost is always upperbounded by a constant amount of time regardless of the input data or the algorithm used.

Example:

- Arithmetic operations: addition, subtraction, multiplication and division
- Comparisons and logical operations
- Assignments, including assignments of pointers when, say, traversing a list or a tree

Order of Growth

Our main concern is about the order of growth.

- Our estimates of time are relative rather than absolute.
- Our estimates of time are machine independent.
- Our estimates of time are about the behavior of the algorithm under investigation on large input instances.

Order of Growth

Our main concern is about the order of growth.

- Our estimates of time are relative rather than absolute.
- Our estimates of time are machine independent.
- Our estimates of time are about the behavior of the algorithm under investigation on large input instances.

So we are measuring the *asymptotic running time* of the algorithms.

The O-Notation

The O-notation provides an *upper bound* of the running time; it may not be indicative of the actual running time of an algorithm.

Definition (O-Notation)

Let $f(n)$ and $g(n)$ be functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $O(g(n))$, written $f(n) = O(g(n))$, if

$$\exists c.\exists n_0.\forall n \geq n_0.f(n) \leq cg(n)$$

Intuitively, f grows no faster than some constant times g.

The Ω-Notation

The Ω-notation provides a *lower bound* of the running time; it may not be indicative of the actual running time of an algorithm.

Definition (Ω-Notation)

Let $f(n)$ and $g(n)$ be functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $\Omega(g(n))$, written $f(n) = \Omega(g(n))$, if

$$\exists c. \exists n_0. \forall n \geq n_0. f(n) \geq cg(n)$$

Clearly $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

The Θ-Notation

The Θ-notation provides an exact picture of the growth rate of the running time of an algorithm.

Definition (Θ-Notation)

Let $f(n)$ and $g(n)$ be functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $\Theta(g(n))$, written $f(n) = \Theta(g(n))$, if both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Clearly $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

Example

Example: $f(n) = 10n^2 + 20n$.

- Since $\forall n \geq 1, f(n) \leq 30n^2, f(n) = O(n^2)$;
- Since $\forall n \geq 1, f(n) \geq n^2, f(n) = \Omega(n^2)$;
- Since $\forall n \geq 1, n^2 \leq f(n) \leq 30n^2, f(n) = \Theta(n^2)$;

Examples

- $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = O(n^k)$.
- $\log n^2 = O(n)$.
- $\log n^k = \Omega(\log n)$.
- $n! = O((n+1)!)$.

Examples

Consider the series $\sum_{j=1}^{n} \log j$. Clearly,

$$\sum_{j=1}^{n} \log j \leq \sum_{j=1}^{n} \log n = n \log n. \text{ Thus } \sum_{j=1}^{n} \log j = O(n \log n)$$

On the other hand,

$$\sum_{j=1}^{n} \log j \geq \sum_{j=1}^{\lfloor n/2 \rfloor} \log(\frac{n}{2}) = \lfloor n/2 \rfloor \log(\frac{n}{2}) = \lfloor n/2 \rfloor \log n - \lfloor n/2 \rfloor$$

That is

$$\sum_{j=1}^{n} \log j = \Omega(n \log n)$$

Examples

- $\log n! = \sum_{j=1}^{n} \log j = \Theta(n \log n)$.
- $2^n = O(n!)$. $(\log 2^n = n)$
- $n! = O(2^{n^2})$. $(\log 2^{n^2} = n^2)$

The o-Notation

Definition (o-Notation)

Let $f(n)$ and $g(n)$ be functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $o(g(n))$, written $f(n) = o(g(n))$, if

$$\forall c . \exists n_0 . \forall n \geq n_0 . f(n) < cg(n)$$

The ω-Notation

Definition (ω-Notation)

Let $f(n)$ and $g(n)$ be functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $\omega(g(n))$, written $f(n) = \omega(g(n))$, if

$$\forall c.\exists n_0.\forall n \geq n_0.f(n) > cg(n)$$

Definition in Terms of Limits

Suppose $\lim_{n\to\infty} f(n)/g(n)$ **exists**.

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$ implies $f(n) = O(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$ implies $f(n) = \Omega(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ implies $f(n) = \Theta(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ implies $f(n) = o(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ implies $f(n) = \omega(g(n))$.

A Helpful Analogy

- $f(n) = O(g(n))$ is similar to $f(n) \le g(n)$.
- $f(n) = o(g(n))$ is similar to $f(n) < g(n)$.
- $f(n) = \Theta(g(n))$ is similar to $f(n) = g(n)$.
- $f(n) = \Omega(g(n))$ is similar to $f(n) \ge g(n)$.
- $f(n) = \omega(g(n))$ is similar to $f(n) > g(n)$.

Complexity Classes

An equivalence relation $\mathcal{R}$ on the set of complexity functions is defined as follows: $f\mathcal{R}g$ if and only if $f(n) = \Theta(g(n))$.

Complexity Classes

An equivalence relation $\mathcal{R}$ on the set of complexity functions is defined as follows: $f\mathcal{R}g$ if and only if $f(n) = \Theta(g(n))$.

A complexity class is an equivalence class of $\mathcal{R}$.

Complexity Classes

An equivalence relation $\mathcal{R}$ on the set of complexity functions is defined as follows: $f\mathcal{R}g$ if and only if $f(n) = \Theta(g(n))$.

A complexity class is an equivalence class of $\mathcal{R}$.

The equivalence classes can be ordered by $\prec$ defined as follows: $f \prec g$ iff $f(n) = o(g(n))$.

Complexity Classes

An equivalence relation $\mathcal{R}$ on the set of complexity functions is defined as follows: $f\mathcal{R}g$ if and only if $f(n) = \Theta(g(n))$.

A complexity class is an equivalence class of $\mathcal{R}$.

The equivalence classes can be ordered by $\prec$ defined as follows: $f \prec g$ iff $f(n) = o(g(n))$.

$1 \prec \log\log n \prec \log n \prec \sqrt{n} \prec n^{\frac{3}{4}} \prec n \prec n\log n \prec n^2 \prec 2^n \prec n! \prec 2^{n^2}$

Outline

Space Complexity

The space complexity is defined to be the number of cells (*work space*)) needed to carry out an algorithm, *excluding the space allocated to hold the input*.

Space Complexity

The space complexity is defined to be the number of cells (*work space*)) needed to carry out an algorithm, *excluding the space allocated to hold the input*.

The exclusion of the input space is to make sense the sublinear space complexity.

Space Complexity

It is clear that the work space of an algorithm can not exceed the running time of the algorithm. That is $S(n) = O(T(n))$.

Space Complexity

It is clear that the work space of an algorithm can not exceed the running time of the algorithm. That is $S(n) = O(T(n))$.

Trade-off between time complexity and space complexity.

Summary

Algorithm	Time Complexity	Space Complexity
LINEARSEARCH		

Summary

Algorithm	Time Complexity	Space Complexity
LINEARSEARCH	$O(n)$	$\Theta(1)$
BINARYSEARCH		

Summary

Algorithm	Time Complexity	Space Complexity
LINEARSEARCH	$O(n)$	$\Theta(1)$
BINARYSEARCH	$O(\log n), \Omega(1)$	$\Theta(1)$
MERGE		

Summary

Algorithm	Time Complexity	Space Complexity
LINEARSEARCH	$O(n)$	$\Theta(1)$
BINARYSEARCH	$O(\log n)$, $\Omega(1)$	$\Theta(1)$
MERGE	$O(n)$, $\Omega(n_1)$	$\Theta(n)$
SELECTIONSORT		

Summary

Algorithm	Time Complexity	Space Complexity
LINEARSEARCH	$O(n)$	$\Theta(1)$
BINARYSEARCH	$O(\log n)$, $\Omega(1)$	$\Theta(1)$
MERGE	$O(n)$, $\Omega(n_1)$	$\Theta(n)$
SELECTIONSORT	$\Theta(n^2)$	$\Theta(1)$
INSERTIONSORT		

Summary

Algorithm	Time Complexity	Space Complexity
LINEARSEARCH	$O(n)$	$\Theta(1)$
BINARYSEARCH	$O(\log n)$, $\Omega(1)$	$\Theta(1)$
MERGE	$O(n)$, $\Omega(n_1)$	$\Theta(n)$
SELECTIONSORT	$\Theta(n^2)$	$\Theta(1)$
INSERTIONSORT	$O(n^2)$, $\Omega(n)$	$\Theta(1)$
BOTTOMUPSORT		

Summary

Algorithm	Time Complexity	Space Complexity
LINEARSEARCH	$O(n)$	$\Theta(1)$
BINARYSEARCH	$O(\log n)$, $\Omega(1)$	$\Theta(1)$
MERGE	$O(n)$, $\Omega(n_1)$	$\Theta(n)$
SELECTIONSORT	$\Theta(n^2)$	$\Theta(1)$
INSERTIONSORT	$O(n^2)$, $\Omega(n)$	$\Theta(1)$
BOTTOMUPSORT	$\Theta(n \log n)$	$\Theta(n)$

Optimal Algorithm

In general, if we can prove that any algorithm to solve problem Π must be $\Omega(f(n))$, then we call any algorithm to solve problem Π in time $O(f(n))$ an *optimal algorithm* for problem Π.

Outline

HOW do we estimate time complexity?

Counting the Iterations

Algorithm 1.7 Count1
Input: $n = 2^k$, for some positive integer k.
Output: *count* = number of times Step 4 is executed.

1. $count \leftarrow 0$;
2. **while** $n \geq 1$
3.   **for** $j \leftarrow 1$ **to** n
4. $count \leftarrow count + 1$
5.   **end for**
6. $n \leftarrow n/2$
7. **end while**
8. **return** *count*

Counting the Iterations

Algorithm 1.7 Count1
Input: $n = 2^k$, for some positive integer k.
Output: *count* = number of times Step 4 is executed.

1. $count \leftarrow 0$;
2. **while** $n \geq 1$
3. **for** $j \leftarrow 1$ **to** n
4. $count \leftarrow count + 1$
5. **end for**
6. $n \leftarrow n/2$
7. **end while**
8. **return** *count*

while is executed $k+1$ times; **for** is executed $n, n/2, \ldots, 1$ times

$$\sum_{j=0}^{k} \frac{n}{2^j} = n \sum_{j=0}^{k} \frac{1}{2^j} = n(2 - \frac{1}{2^k}) = 2n - 1 = \Theta(n)$$

Counting the Iterations

Algorithm 1.8 Count2

Input: A positive integer n.

Output: *count* = number of times Step 5 is executed.

1. $count \leftarrow 0$;
2. **for** $i \leftarrow 1$ **to** n
3. $m \leftarrow \lfloor n/i \rfloor$
4.   **for** $j \leftarrow 1$ **to** m
5. $count \leftarrow count + 1$
6.   **end for**
7. **end for**
8. **return** *count*

Counting the Iterations

Algorithm 1.8 Count2
Input: A positive integer n.
Output: *count* = number of times Step 5 is executed.

1. $count \leftarrow 0$;
2. **for** $i \leftarrow 1$ **to** n
3. $\quad m \leftarrow \lfloor n/i \rfloor$
4. $\quad$ **for** $j \leftarrow 1$ **to** m
5. $\qquad count \leftarrow count + 1$
6. $\quad$ **end for**
7. **end for**
8. **return** *count*

The inner **for** is executed $n, \lfloor n/2 \rfloor, \lfloor n/3 \rfloor, \ldots, \lfloor n/n \rfloor$ times

$$\Theta(n \log n) = \sum_{i=1}^{n} \left(\frac{n}{i} - 1\right) \leq \sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor \leq \sum_{i=1}^{n} \frac{n}{i} = \Theta(n \log n)$$

Counting the Iterations

Algorithm 1.9 Count3
Input: $n = 2^{2^k}$, k is a positive integer.
Output: *count* = number of times Step 6 is executed.

1. $count \leftarrow 0$;
2. **for** $i \leftarrow 1$ **to** n
3. $\quad j \leftarrow 2$;
4. $\quad$ **while** $j \leq n$
5. $\qquad j \leftarrow j^2$;
6. $\qquad count \leftarrow count + 1$
7. $\quad$ **end while**
8. **end for**
9. **return** *count*

Counting the Iterations

For each value of i, the **while** loop will be executed when $j = 2, 2^2, 2^4, \cdots, 2^{2^k}$.

That is, it will be executed when $j = 2^{2^0}, 2^{2^1}, 2^{2^2}, \cdots, 2^{2^k}$.

Thus, the number of iterations for **while** loop is $k + 1 = \log\log n + 1$ for each iteration of **for** loop.

The total output is $n(\log\log n + 1) = \Theta(n \log\log n)$.

Counting the Iterations

Algorithm 1.10 PSUM

Input: $n = k^2$, k is a positive integer.

Output: $\sum_{i=1}^{j} i$ for each perfect square j between 1 and n.

1. $k \leftarrow \sqrt{n}$;
2. **for** $j \leftarrow 1$ **to** k
3. $\quad sum[j] \leftarrow 0$;
4. $\quad$ **for** $i \leftarrow 1$ **to** j^2
5. $\quad\quad sum[j] \leftarrow sum[j] + i$;
6. $\quad$ **end for**
7. **end for**
8. **return** $sum[1 \cdots k]$

Counting the Iterations

Assume that $\sqrt{n}$ can be computed in $O(1)$ time.

The outer and inner **for** loop are executed $k = \sqrt{n}$ and j^2 times respectively.

Thus, the number of iterations for inner **for** loop is

$$\sum_{j=1}^{k}\sum_{i=1}^{j^2} 1 = \sum_{j=1}^{k} j^2 = \frac{k(k+1)(2k+1)}{6} = \Theta(k^3) = \Theta(n^{1.5}).$$

The total output is $\Theta(n^{1.5})$.

Counting the Frequency of Basic Operations

Definition

An elementary operation in an algorithm is called a *basic operation* if it is of highest frequency to within a constant factor among all other elementary operations.

Method of Choice

- When analyzing searching and sorting algorithms, we may choose the element comparison operation if it is an elementary operation.
- In matrix multiplication algorithms, we select the operation of scalar multiplication.
- In traversing a linked list, we may select the "operation" of setting or updating a pointer.
- In graph traversals, we may choose the "action" of visiting a node, and count the number of nodes visited.

Master theorem

If

$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

for some constants $a > 0$, $b > 1$, and $d \geq 0$,

Master theorem

If

$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a. \end{cases}$$

Analysis for MERGESORT

The recurrence relation:

$$T(n) = 2T(n/2) + O(n);$$

Analysis for MERGESORT

The recurrence relation:

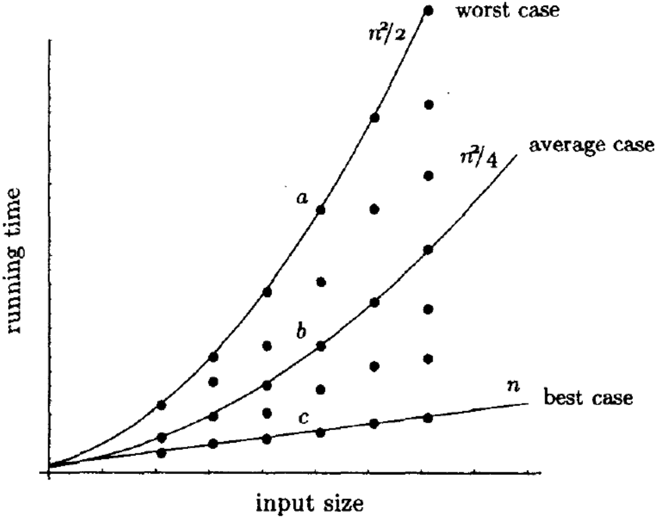
$$T(n) = 2T(n/2) + O(n);$$

By Master Theorem

$$T(n) = O(n \log n).$$

Outline

Performance of INSERTIONSORT



Worst Case Analysis

Consider the following algorithm:

1. **if** n is odd **then** $k \leftarrow \text{BinarySearch}(A, x)$
2. **else** $k \leftarrow \text{LinearSearch}(A, x)$

In the worst case, the running time is $\Omega(\log(n))$ and $O(n)$.

Average Case Analysis

Take Algorithm InsertionSort for instance. Two assumptions:

- $A[1..n]$ contains the numbers 1 through n.
- All $n!$ permutations are equally likely.

The number of comparisons for inserting element $A[i]$ in its proper position, say j, is *on average* the following

$$\frac{i-1}{i} + \sum_{j=2}^{i} \frac{i-j+1}{i} = \frac{i-1}{i} + \sum_{j=1}^{i-1} \frac{j}{i} = \frac{i}{2} - \frac{1}{i} + \frac{1}{2}$$

The *average* number of comparisons performed by Algorithm InsertionSort is

$$\sum_{i=2}^{n} \left(\frac{i}{2} - \frac{1}{i} + \frac{1}{2}\right) = \frac{n^2}{4} + \frac{3n}{4} - \sum_{i=1}^{n} \frac{1}{i}$$

Amortized Analysis

In amortized analysis, we average out the time taken by the operation throughout the execution of the algorithm, and refer to this average as the *amortized running time* of that operation.

Amortized Analysis

In amortized analysis, we average out the time taken by the operation throughout the execution of the algorithm, and refer to this average as the *amortized running time* of that operation.

Amortized analysis guarantees the average cost of the operation, and thus the algorithm, *in the worst case*.

Amortized Analysis

In amortized analysis, we average out the time taken by the operation throughout the execution of the algorithm, and refer to this average as the *amortized running time* of that operation.

Amortized analysis guarantees the average cost of the operation, and thus the algorithm, *in the worst case*.

This is to be contrasted with the average time analysis in which the average is taken over all instances of the same size. Moreover, unlike the average case analysis, no assumptions about the probability distribution of the input are needed.

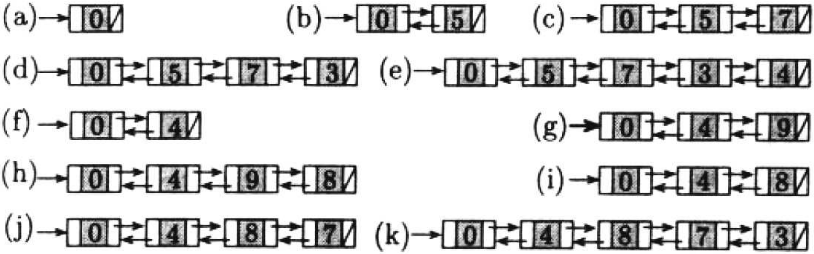
Amortized Analysis

Consider the following algorithm:

1. **for** $j \leftarrow 1$ **to** n
2. $x \leftarrow A[j]$
3. Append x to the list
4. **if** x is even **then**
5. **while** $pred(x)$ is odd **do** delete $pred(x)$
6. **end if**
7. **end for**

An Example

5	7	3	4	9	8	7	3



Analysis

Worst Case Analysis: If no input numbers are even, or if all even numbers are at the beginning, then no elements are deleted, and hence each iteration of the **for** loop takes constant time. However, if the input has $n - 1$ odd integers followed by one even integer, then the number of deletions is $n - 1$, and the number of **while** loops is $n - 1$. The overall running time is $O(n^2)$.

Amortized Analysis: The total number of elementary operations of insertions and deletions is between n and $2n - 1$. So the time complexity is $\Theta(n)$. It follows that the time used to delete each element is $O(1)$ **amortized** time.

Input Size and Problem Instance

Suppose that the following integer

$$2^{1024} - 1$$

is a legitimate input of an algorithm. What is the *size* of the input?

Input Size and Problem Instance

Algorithm 1.9 FIRST
Input: A positive integer n and an array $A[1..n]$ with $A[j] = j$ for $1 \leq j \leq n$.
Output: $\sum_{j=1}^{n} A[j]$.

1. $sum \leftarrow 0$;
2. **for** $j \leftarrow 1$ **to** n
3. $\quad sum \leftarrow sum + A[j]$
4. **end for**
5. **return** sum

The input size is n. The time complexity is $O(n)$. It is linear time.

Input Size and Problem Instance

Algorithm 1.10 SECOND
Input: A positive integer n.
Output: $\sum_{j=1}^{n} j$.

1. $sum \leftarrow 0$;
2. **for** $j \leftarrow 1$ **to** n
3. $\quad sum \leftarrow sum + j$
4. **end for**
5. **return** sum

The input size is $k = \lfloor \log n \rfloor + 1$. The time complexity is $O(2^k)$. It is exponential time.

Commonly Used Measures

- In sorting and searching problems, we use the number of entries in the array or list as the input size.
- In graph algorithms, the input size usually refers to the number of vertices or edges in the graph, or both.
- In computational geometry, the size of input is usually expressed in terms of the number of points, vertices, edges, line segments, polygons, etc.
- In matrix operations, the input size is commonly taken to be the dimensions of the input matrices.
- In number theory algorithms and cryptography, the number of bits in the input is usually chosen to denote its length. The number of words used to represent a single number may also be chosen as well, as each word consists of a fixed number of bits.