Algorithm Design and Analysis

Jing Liu Kai Wu

吴 凯

- 副教授/硕士生导师
- 华山菁英副教授

联系方式：
kwu@xidian.edu.cn
主页链接：
https://web.xidian.edu.cn/kwu/index.html

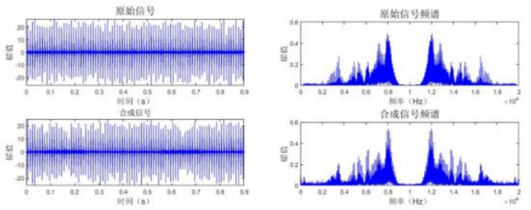
1、个人履历
2015年与2020年分别在西安电子科技大学获得智能科学与技术专业学士学位、电路与系统博士学位。2020年留校任教至今，现为华山菁英副教授。

2、个人荣誉
陕西省优秀博士学位论文

3、研究方向
AI for Network Sciences, Automated Evolutionary Computation, Deep Learning for Time Series Analysis, Adversarial Machine Learning

4、工作业绩
以第一作者/通讯作者在IEEE TNNLS、IEEE TFS、IEEE TEVC、IEEE TCy、IEEE CIM、中国科学: 技术科学等国内外刊物发表论文20余篇，博士学位论文入选陕西省优秀博士学位论文、西安电子科技大学2022年校优秀博士学位论文，获第五届智能优化与调度学术会议优秀博士论文奖。获批国家自然科学基金青年项目、陕西省科学技术协会青年人才托举计划项目、华山人才基金和之江实验室国际青年人才基金支持。参与国家自然科学基金面上项目、科技部新一代人工智能重点研发计划等。受邀担任2020-2023 IEEE SSCI多智能体协同与优化研讨会的主席。

5. 代表性产业化应用案例：多工况轴承振动信号增强
技术优势：
（1） 解决了多工况轴承振动信号增强精度低的问题
（2）解决了未知工况的轴承振动信号增强问题
（3）中，晚期多故障类型的多场景覆盖
应用场景：
适用传感器获取信号的各种场景、如电动汽车、工业电机、轨道交通、航空航天



本科生招聘

团队目前有以下方向需要攻关，欢迎对以下方向感兴趣的本科生同学联系

•方向一：时间序列生成：数据永远是稀缺的，对时间序列而言更是如此
内容：参与高效可用的时间序列生成集成算法平台构建、团队在该领域idea的验证、论文写作等

•方向二：优化的自动化：如何减少人力成本，实现优化的自动化，LLM agent可能是一条有趣的路径
内容：参与高效可用的优化自动化集成平台构建、团队在该领域idea的验证、论文写作等

•方向三：图基础模型：图的语义是什么？这可能关乎着graph foundation model能否像 NLP和CV领域的基础模型一样有效
内容：参与高效可用的图基础模型集成平台构建、团队在该领域idea的验证、论文写作等

要求

希望你：
有空闲的时间和精力；
提前或多或少对上述领域有了解或有自己的理解；
自驱力强，能够坚持进行探究；
有一定的代码能力，或有毅力去独立学习、编写相关代码。

你将收获：
科研内容指导与讨论；
相对自主地尝试并不复杂的科研；
如有论文等成果产出，你将是第一作者。

竞赛指导、科研项目实习等

- Evol Group

欢迎大家加入**EvolGroup**

联系方式：**kwu@xidian.edu.cn**

Algorithm Design and Analysis

Lecturer: Kai Wu

Email: kwu@xidian.edu.cn

Homepage: https://web.xidian.edu.cn/kwu/

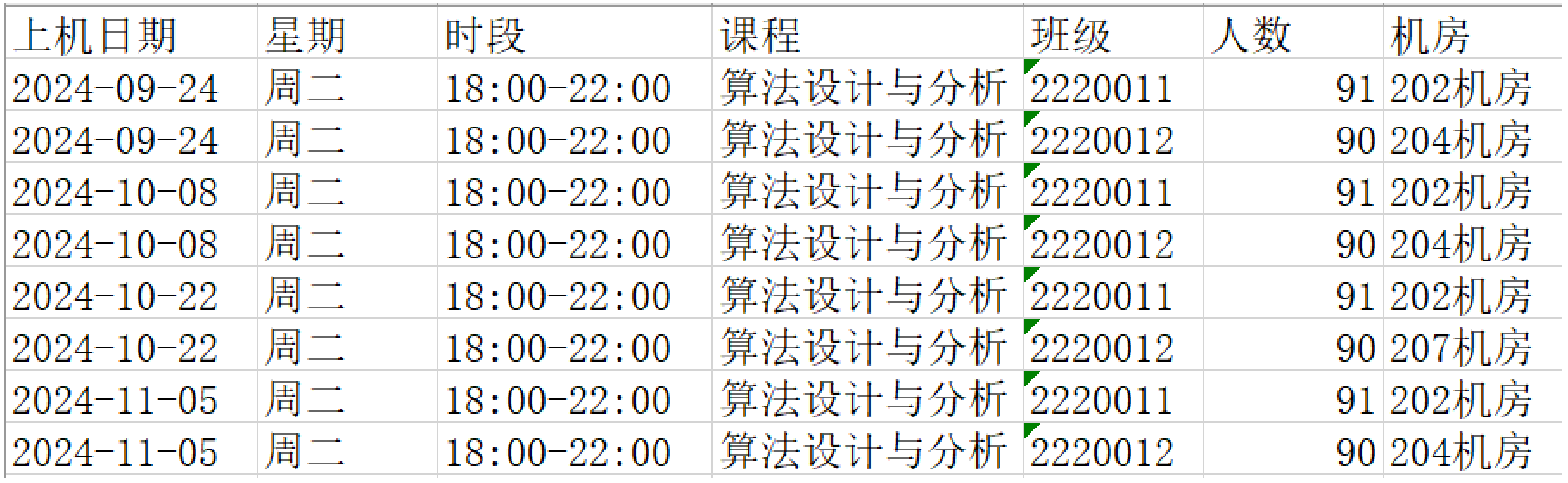
Textbook

- M. H. Alsuwaiyel, Algorithms design techniques and Analysis, Publishing House of Electronics Industry

- M. H. Alsuwaiyel著，吴伟昶，方世昌等译，算法设计技巧与分析，电子工业出版社

Practicing Classes

- 4 times, 4 hours each time in the lab



上机日期	星期	时段	课程	班级	人数	机房
2024-09-24	周二	18:00-22:00	算法设计与分析	2220011	91	202机房
2024-09-24	周二	18:00-22:00	算法设计与分析	2220012	90	204机房
2024-10-08	周二	18:00-22:00	算法设计与分析	2220011	91	202机房
2024-10-08	周二	18:00-22:00	算法设计与分析	2220012	90	204机房
2024-10-22	周二	18:00-22:00	算法设计与分析	2220011	91	202机房
2024-10-22	周二	18:00-22:00	算法设计与分析	2220012	90	207机房
2024-11-05	周二	18:00-22:00	算法设计与分析	2220011	91	202机房
2024-11-05	周二	18:00-22:00	算法设计与分析	2220012	90	204机房

Grading

- Final exam: 70%
- Others: 30%

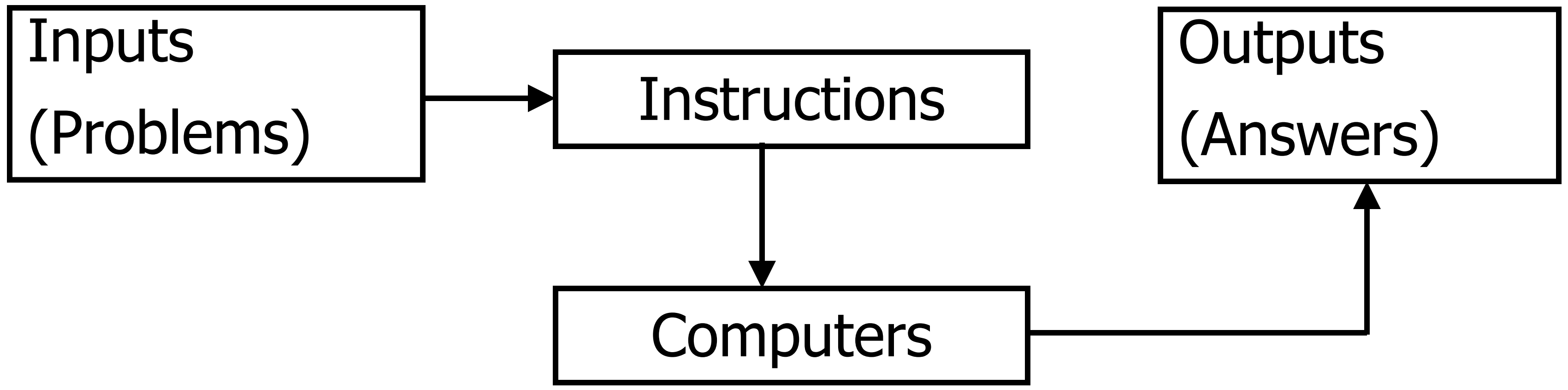
Four practices, each 5% + Final presentation (individual), 10%

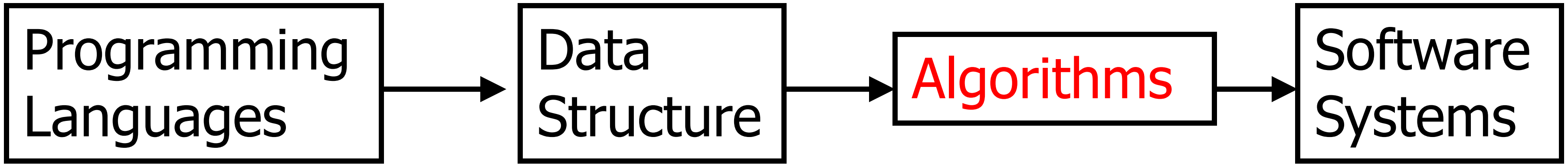
Content

- Chapter 1 Basic Concepts in Algorithmic Analysis
- Chapter 5 Induction
- Chapter 6 Divide and Conquer
- Chapter 7 Dynamic Programming
- Chapter 8 The Greedy Approach
- Chapter 9 Graph Traversal
- Chapter 13 Backtracking
- Chapter 14 Randomized Algorithms
- Chapter 15 Approximation Algorithms
- Chapter 16 Network Flow
- Chapter 17 Matching

What's Algorithms?

- 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* if such an output exists or else obtain nothing at all if there is no output for that particular input through a systematic execution of the instructions.


Inputs (Problems)
Instructions
Computers
Outputs (Answers)


Programming Languages
Data Structure
Algorithms
Software Systems

Binary Search

- Let $A[1\ldots n]$ be a sequence of n elements. Consider the problem of determining whether a given element x is in A.

Binary Search

Example:

$A[1\ldots14]$=1 4 5 7 8 9 10 12 15 22 23 27 32 35

x=22

Does X exist in A? How many comparisons do you need to give the answer?

$A[1\ldots14]$=1 4 5 7 8 9 10 12 15 22 23 27 32 35

x=22

Binary Search

- **Inputs**: (1) An array $A[1\ldots n]$ of n elements sorted in nondecreasing order; (2) x;
- **Output**: j if $x=A[j]$, and 0 otherwise;

```
1. $low \leftarrow 1$; $high \leftarrow n$; $j \leftarrow 0$;
2. while ($low \leq high$) and ($j=0$)
3.      $mid \leftarrow \lfloor (low+high)/2 \rfloor$;
4.      if $x=A[mid]$ then $j \leftarrow mid$;
5.      else if $x<A[mid]$ then $high \leftarrow mid-1$;
6.      else $low \leftarrow mid+1$;
7. end while;
8. return $j$;
```

Binary Search

- **What's the performance of the algorithm Binary Search on a sorted array of size *n*?**
  - ✓ What's the minimum number of comparisons we need and in which case it will happen?
  - ✓ What's the maximum number of comparisons we need and in which case it will happen?

Binary Search

- The number of comparisons performed by the algorithm Binary Search on a sorted array of size n is at most $\lfloor \log n \rfloor + 1$.

Binary Search

- Can you implement the Binary Search in a certain kind of computer language?

Merging Two Sorted Lists

- Suppose we have an array $A[1\ldots m]$ and three indices p, q, and r, with $1 \leq p \leq q < r \leq m$, such that both the subarrays $A[p\ldots q]$ and $A[q+1\ldots r]$ are individually sorted in nondecreasing order. We want to resort the elements in A so that the elements in the subarray $A[p\ldots r]$ are sorted in nondecreasing order.

Merging Two Sorted Lists

Example:

$A[1…7]$=5 8 11 4 8 10 12

p=1, q=3, r=7

$A[1…3]$ and $A[4…7]$ are already sorted nondecreasingly, please sort A nondecreasingly? How many comparisons do you need to give the answer?

Merging Two Sorted Lists

- **Inputs**: (1) $A[1\ldots m]$; (2) p, q, and r, and $1 \le p \le q < r \le m$, such that both $A[p\ldots q]$ and $A[q+1\ldots r]$ are sorted individually in nondecreasing order;

- **Output:** $A[p\ldots r]$ contains the result of merging the two subarrays $A[p\ldots q]$ and $A[q+1\ldots r]$;

Merging Two Sorted Lists

```
1. $s \leftarrow p$; $t \leftarrow q+1$; $k \leftarrow p$;
2. while $s \le q$ and $t \le r$
3.     if $A[s] \le A[t]$ 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 $B[k \ldots r] \leftarrow A[t \ldots r]$
13. else $B[k \ldots r] \leftarrow A[s \ldots q]$
14. end if
15. $A[p \ldots r] \leftarrow B[p \ldots r]$
```

$A[1 \ldots 7] = 5 \quad 8 \quad 11 \quad 4 \quad 8 \quad 10 \quad 12$

$p=1$, $q=3$, $r=7$

Merging Two Sorted Lists

- **What's the performance of the algorithm Merging Two Sorted Lists?**
  - ✓ What's the minimum number of comparisons we need and in which case it will happen?
  - ✓ What's the maximum number of comparisons we need and in which case it will happen?
  - ✓ What's the minimum number of element assignments we need and in which case it will happen?
  - ✓ What's the maximum number of element assignments we need and in which case it will happen?

Merging Two Sorted Lists

```
1. $s \leftarrow p$; $t \leftarrow q+1$; $k \leftarrow p$;
2. while $s \le q$ and $t \le r$
3.     if $A[s] \le A[t]$ 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 $B[k \ldots r] \leftarrow A[t \ldots r]$
13. else $B[k \ldots r] \leftarrow A[s \ldots q]$
14. end if
15. $A[p \ldots r] \leftarrow B[p \ldots r]$
```

$A[1\ldots7]=5\ \ 8\ \ 11\ \ 4\ \ 8\ \ 10\ \ 12$

$p=1$, $q=3$, $r=7$

Merging Two Sorted Lists

- The number of element comparisons performed by Algorithm MERGE to merge two nonempty arrays of sizes n_1 and n_2, respectively, where $n_1 \leq n_2$, into one sorted array of size $n = n_1 + n_2$ is between n_1 and $n-1$.

Merging Two Sorted Lists

```
1. $s \leftarrow p$; $t \leftarrow q+1$; $k \leftarrow p$;
2. while $s \le q$ and $t \le r$
3.     if $A[s] \le A[t]$ 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 $B[k \ldots r] \leftarrow A[t \ldots r]$
13. else $B[k \ldots r] \leftarrow A[s \ldots q]$
14. end if
15. $A[p \ldots r] \leftarrow B[p \ldots r]$
```

$A[1\ldots7]=5\ \ 8\ \ 11\ \ 4\ \ 8\ \ 10\ \ 12$

$p=1$, $q=3$, $r=7$

Merging Two Sorted Lists

- The number of element assignments performed by Algorithm MERGE to merge two arrays into one sorted array of size n is exactly $2n$.

Problems and Algorithms

- Each algorithm is designed to solve one problem, but for one problem, we can design many different algorithms.
  - ✓ What's the difference of these algorithms?
  - ✓ Why we design different algorithms for the same problem?

Example Problem: Sort

- ✓ Selection Sort
- ✓ Insertion Sort
- ✓ Bottom-Up Merge Sorting

Selection Sort

- Let $A[1\ldots n]$ be an array of n elements.
- A simple and straightforward algorithm to sort the entries in A works as follows.
- First, we find the minimum element and store it in $A[1]$.
- Next, we find the minimum of the remaining n-1 elements and store it in $A[2]$.
- We continue this way until the second largest element is stored in $A[n-1]$.

Selection Sort

- **Input**: $A[1\ldots n]$;
- **Output**: $A[1\ldots n]$ sorted in nondecreasing order;

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

Selection Sort

- **What's the performance of the algorithm Selection Sort?**
  - ✓ What's the minimum number of comparisons we need and in which case it will happen?
  - ✓ What's the maximum number of comparisons we need and in which case it will happen?
  - ✓ What's the minimum number of element assignments?
  - ✓ What's the maximum number of element assignments?

Selection Sort

- The number of element comparisons performed by Algorithm SELECTIONSORT to sort an array with *n* elements is $n(n-1)/2$.

Selection Sort

- The number of element assignments performed by Algorithm SELECTIONSORT to sort an array with n elements is between 0 and $3(n-1)$.

Test

- A = [1 6 4 3 9 8 2]. Sort A using SELECTIONSORT and give the number of comparisons and assignments?

Insertion Sort

- We begin with the subarray of size 1, *A*[1], which is already sorted.
- Next, *A*[2] is inserted before or after *A*[1] depending on whether it is smaller than *A*[1] or not.
- Continuing this way, in the *i* th iteration, *A*[*i*] is inserted in its proper position in the sorted subarray *A*[1 … *i*-1].
- This is done by scanning the elements from index *i*-1 down to 1, each time comparing *A*[*i*] with the element at the current position.

Insertion Sort

- In each iteration of the scan, an element is shifted one position up to a higher index. This process of scanning, performing the comparison and shifting continues until an element less than or equal to $A[i]$ is found, or when all the sorted sequence so far is exhausted.
- At this point, $A[i]$ is inserted in its proper position, and the process of inserting element $A[i]$ in its proper place is complete.

Insertion Sort

Example:

$A[1 \ldots 4]$=9 4 5 2

Insertion Sort

- **Input**: An array $A[1\ldots n]$ of n elements;
- **Output**: $A[1\ldots n]$ sorted in nondecreasing order;

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

Insertion Sort

- **What's the performance of the algorithm Insertion Sort?**
  - ✓ What's the minimum number of comparisons we need and in which case it will happen?
  - ✓ What's the maximum number of comparisons we need and in which case it will happen?
  - ✓ What's the minimum number of element assignments?
  - ✓ What's the maximum number of element assignments?

Insertion Sort

- The number of element comparisons performed by Algorithm INSERTIONSORT is between n-1 and $n(n$-1)/2.

Insertion Sort

- The number of element assignments performed by Algorithm INSERTIONSORT is equal to the number of element comparisons plus n-1.

Test

- A = [1 6 4 3 9 8 2]. Sort A using INSERTIONSORT and give the number of comparisons and assignments?

Bottom-Up Merge Sorting

- Let A be an array of n elements that is to be sorted.
- We first merge $\lfloor n/2 \rfloor$ consecutive pairs of elements to yield $\lfloor n/2 \rfloor$ sorted sequences of size 2.
- If there is one remaining element, then it is passed to the next iteration.
- Next, we merge $\lfloor n/4 \rfloor$ pairs of consecutive 2-element sequences to yield $\lfloor n/4 \rfloor$ sorted sequences of size 4.
- If there are one or two remaining elements, then they are passed to the next iteration.
- If there are three elements left, then two (sorted) elements are merged with one element to form a 3-element sorted sequence.

Bottom-Up Merge Sorting

- Continuing this way, in the jth iteration, we merge $\lfloor n/2^j \rfloor$ pairs of sorted sequences of size 2^{j-1} to yield $\lfloor n/2^j \rfloor$ sorted sequences of size 2^j.
- If there are k remaining elements, where $1 \leq k \leq 2^{j-1}$, then they are passed to the next iteration.
- If there are k remaining elements, where $2^{j-1} < k < 2^j$, then these are merged to form a sorted sequence of size k.

Bottom-Up Merge Sorting

- **Input**: An array $A[1\ldots n]$ of n elements;
- **Output**: $A[1\ldots n]$ sorted in nondecreasing order;

```
1. $t \leftarrow 1$;
2. while $t < n$
3.     $s \leftarrow t$; $t \leftarrow 2s$; $i \leftarrow 0$;
4.     while $i + t \le n$
5.         MERGE($A$, $i+1$, $i+s$, $i+t$);
6.         $i \leftarrow i+t$;
7.     end while;
8.     if $i+s<n$ then MERGE($A$, $i+1$, $i+s$, $n$);
9. end while;
```

Bottom-Up Merge Sorting

Example:

$A[1 \ldots 8]$=9 4 5 2 1 7 4 6

$A[1 \ldots 11]$=6 10 9 5 3 11 4 8 1 2 7

Bottom-Up Merge Sorting

Example:

```
1. $t \leftarrow 1$;
2. while $t < n$
3.     $s \leftarrow t$; $t \leftarrow 2s$; $i \leftarrow 0$;
4.     while $i+t \le n$
5.         MERGE($A$, $i+1$, $i+s$, $i+t$);
6.         $i \leftarrow i+t$;
7.     end while;
8.     if $i+s<n$ then MERGE($A$, $i+1$, $i+s$, $n$);
9. end while;
```

Bottom-Up Merge Sorting

- **What's the performance of the algorithm Bottom-Up Merge Sorting?**
  - ✓ What's the minimum number of comparisons we need and in which case it will happen?
  - ✓ What's the maximum number of comparisons we need and in which case it will happen?
  - ✓ What's the minimum number of element assignments?
  - ✓ What's the maximum number of element assignments?

Bottom-Up Merge Sorting

- The total number of element comparisons performed by Algorithm BOTTOMUPSORT to sort an array of n element, where n is a power of 2, is between $(n\log n)/2$ and $n\log n-n+1$.

Bottom-Up Merge Sorting

- The total number of element assignments performed by Algorithm BOTTOMUPSORT to sort an array of *n* element, where *n* is a power of 2, is exactly 2*n*log *n*.

Time Complexity

- Selection Sort: $n(n\text{-}1)/2$
- Merge Sort: $n\log n - n + 1$

If each comparison needs 10^{-6} second,

- n=128: Merge=0.0008 seconds

  Selection=0.008 seconds

- $n=2^{20}$: Merge=20 seconds

  Selection=6.4 days

Time Complexity

When analyzing the running time,

(1) We usually compare its behavior with another algorithm that solves the same problem.

(2) It is desirable for an algorithm to be not only machine independent, but also capable of being expressed in any language.

(3) It should be technology independent, that is, we want our measure of the running time of an algorithm to survive technological advances.

(4) Our main concern is not in small input sizes; we are mostly concerned with the behavior of the algorithm on large input instances.

Time Complexity

Therefore, usually only ***Elementary Operations*** are used to evaluate the time complexity:

- **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.

Time Complexity

Examples of elementary operations:

(1) Arithmetic operations: addition, subtraction, multiplication and division

(2) Comparisons and logical operations

(3) Assignments, including assignments of pointers

Time Complexity

- Usually, we care about how the elementary operations increase with the size of input, namely the rate of growth or the order of growth of the running time. This can be expressed by a function, for example:

$$f(n) = n^2 \log n + 10n^2 + n$$

- Once we dispose of lower order terms and leading constants from a function that expresses the running time of an algorithm, we say that we are measuring the ***asymptotic running time*** of the algorithm, namely ***time complexity***.

Time Complexity

- Functions that are widely used to represent the running times of algorithms:

- (1) Sublinear: n^c, $n^c\log^k n$, $0<c<1$
- (2) Linear: cn
- (3) Logarithmic: $\log^k n$
- (4) Subquadratic: $n\log n$, $n^{1.5}$
- (5) Quadratic: cn^2
- (6) Cubic: cn^3

Time Complexity

In order to formalize the notions of time complexity, special mathematical notations have been widely used.

(1) **O-notation**: An upper bound on the running time. The running time of an algorithm is $O(g(n))$, if whenever the input size is equal to or exceeds some threshold n_0, its running time can be bounded above by some positive constant c times $g(n)$.

(2) **Ω-notation**: A lower bound on the running time. The running time of an algorithm is $\Omega(g(n))$, if whenever the input size is equal to or exceeds some threshold n_0, its running time can be bounded below by some positive constant c times $g(n)$.

Time Complexity

(3) **Θ-notation**: An exact bound. The running time of an algorithm is of order $\Theta(g(n))$ if whenever the input size is equal to or exceeds some threshold n_0, its running time can be bounded below by $c_1 g(n)$ and above by $c_2 g(n)$, where $0 < c_1 \leq c_2$.

Time Complexity

(1) $f(n)$ is $\Omega(g(n))$ if and only if $g(n)$ is $O(f(n))$

(2) $f(n)=\Theta(g(n))$ if and only if $f(n)=O(g(n))$ and $f(n)= \Omega(g(n))$

Time Complexity

- It may be helpful to think of O as similar to $\leq$, Ω as similar to $\geq$, and Θ as similar to $=$.
- Don't confuse the exact relations with the asymptotic notations.
- $100n \geq n$, but $\exists c$, s.t. when $n > n_0$, $100n \leq cn \Rightarrow 100n = O(n)$
- $n \leq 100n$, but $\exists c$, s.t. when $n > n_0$, $n \geq c100n \Rightarrow n = \Omega(100n)$, such as $c \leq 0.01$
- $n \neq 100n$, but $\exists c_1$, c_2, s.t. when $n > n_0$, $c_1 100n \leq n \leq c_2 100n \Rightarrow n = \Theta(100n)$. Such as $c_1 \leq 0.01$, $c_2 \geq 0.01$

Time Complexity

- How to use the notations O, Ω, Θ to represent the time complexity of the three previous sorting algorithms?

Time Complexity

- How to use the notations O, Ω, Θ to represent the following functions?

✓ Any constant functions

✓ $\log n^2$

✓ $\log n^k$

✓ $\sum_{j=1}^{n} \log j$

✓ $\log n!$

Space Complexity

- We define the space used by an algorithm to be the number of memory cells (or words) needed to carry out the computational steps required to solve an instance of the problem excluding the space allocated to hold the input. That is, it is only the work space required by the algorithm.

Space Complexity

- The work space cannot exceed the running time of an algorithm, as writing into each memory cell requires at least a constant amount of time.

- Let $T(n)$ and $S(n)$ denote, respectively, the time and space complexities of an algorithm, then

$$S(n) = O(T(n))$$

Space Complexity

- Selection Sort and Insertion Sort: $\Theta(1)$
- Merge: $\Theta(n)$
- Merge Sort: $\Theta(n)$

How to Estimate the Running Time of an Algorithm

(1) Counting the number of iterations:

(n=2^k)

```
$count \leftarrow 0$
while $n \geq 1$
    for $j \leftarrow 1$ to $n$
        $count \leftarrow count + 1$
    end for
    $n \leftarrow n/2$
end while
```

How to Estimate the Running Time of an Algorithm

(2) Counting the frequency of basic operations

(3) Using recurrence relations: In recursive algorithms, a formula bounding the running time is usually given in the form of a recurrence relation; that is, a function whose definition contains the function itself.

Example: Analyze the performance of Binary Search.

Worst Case and Average Case Analysis

- Insertion Sort (Increasing): The performance is related not only to the input size, but also to the order of input.

Input Order	**Performance**
✓ Decreasing	✓ Worst
✓ Random	✓ Average
✓ Increasing	✓ Best

Worst Case and Average Case Analysis

- **Worst case analysis**: Select the maximum cost among all possible inputs.

- **Average case analysis**: It is necessary to know the probabilities of all input occurrences, i.e., it requires prior knowledge of the input distribution.

Example: Insertion Sort

Amortized Analysis

- Consider an algorithm in which an operation is executed repeatedly with the property that its running time fluctuates throughout the execution of the algorithm.

- If this operation takes a large amount of time ***occasionally*** and runs much faster ***most of the time***, then this is an indication that amortized analysis should be employed.

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.
- No assumptions about the probability distribution of the input are needed.

Amortized Analysis

Example: We have a *doubly linked list* that initially consists of one node which contains the integer 0. We have as input an array $A[1\ldots n]$ of n positive integers that are to be processed in the following way. If the current integer x is odd, then append x to the list. If it is even, then first append x and then remove all odd elements before x in the lists.

$A[1\ldots 8]=5\ \ 7\ \ 3\ \ 4\ \ 9\ \ 8\ \ 7\ \ 3$

Input Size and Problem Instance

A measure of the performance of an algorithm is usually a function of its input: its size, order, distribution, etc. The most prominent of these, which is of interest to us here, is the ***input size***. When discussing a problem, as opposed to an algorithm, we usually talk of a ***problem instance***.

- ***Input Size***: belongs to the practical part of algorithm analysis.

- ***Problem Instance***: translates to input in the context of an algorithm that solves that problems.

Input Size and Problem Instance

For example, we call an array *A* of *n* integers an instance of the sort problem. At the same time, in the context of discussing algorithm, we refer to this array as an input to the algorithm.