Algorithms and Data Structures II

Lecture 1:

Algorithms and Their Complexity

https://elms.u-aizu.ac.jp/

Consider

- To solve a problem by a computer, an algorithm is needed.
- Given an algorithm for the problem, we want to know the efficiency of the algorithm.
- We are most interested in how much time and how much memory space the algorithm takes to solve the problem.

Consider (cntd. 1)

- The computation time of an algorithm depends on the number of computation steps of the algorithm and the computer used.

- To evaluate the efficiency of algorithms, it is ideal to use an unique computer to measure their computation time.

- The number of computation steps of an algorithm represents its computation time.

Consider (cntd. 2)

- The computation time of an algorithm for a problem depends on the size of the problem.
- How the computation time of the algorithm grows when the size of the problem increases is important.

Consider (cntd. 3)

- The size of a problem is denoted by an integer n, which is a measure of the quantity of input data.
  - The size of a matrix multiplication problem might be the largest dimension of the matrices.
  - The size of a sorting problem might be the number of data to be sorted.
  - The size of a graph problem might be the number of vertices or edges.

Consider (cntd. 4)

- The computation time needed by an algorithm expressed as a function of the size of a problem is called time complexity of the algorithm. Analogous definition can be made for space complexity.

Consider (cntd. 5)

- Given an algorithm for a problem of size n, it is important to find the time complexity and how the time complexity grows when n increases.

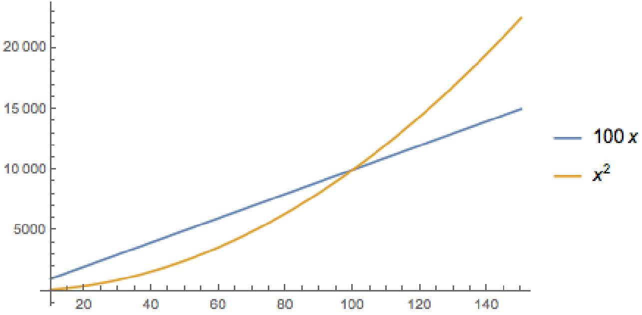
- It is the growth rate of the time complexity (space complexity) of an algorithm which ultimately determines the size of problems that can be solved by the algorithm.

Definitions of time efficiency

- Def. 1: $t(n) = O(f(n))$ if $\exists$ constants $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0 \;\; t(n) \leq cf(n)$.
- Def. 2: $t(n) = \Omega(f(n))$ if $\exists$ constants $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0 \;\; t(n) \geq cf(n)$.
- Def. 3: $t(n) = o(f(n))$ if $\lim_{n\to\infty} t(n)/f(n) = 0$.
- Def. 4: $t(n) = \Theta(f(n))$ if $\exists$ constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $\forall n \leq n_0 \;\; c_1 f(n) \leq t(n) \leq c_2 f(n)$.
- These definitions are used to establish a relative order among functions. To compare the time efficiency of algorithms, we compare the relative rates of growth of their time complexities.

For example,

- Given two functions $f_1 = 100n$ and $f_2 = n^2$, although $100n$ is larger than n^2 for small n, n^2 grows at a fast rate and becomes larger than $100n$ for all $n > 100$.



Def. 1: $t(n) = O(f(n))$ **if** $\exists$ **constants** $c > 0$ **and** $n_0 > 0$ **such that** $\forall n \geq n_0 \quad t(n) \leq cf(n)$**.**

- Def. 1 says that if constant factors are ignored $f(n)$ is at least as large as $t(n)$.
- $t(n) = O(f(n))$ means that the growth rate of $t(n)$ is smaller than or equal to the growth rate of $f(n)$.
- $O(\ldots)$ is read as " order ... " or " Big-Oh .... "

Def. 2: $t(n) = \Omega(f(n))$ **if** $\exists$ **constants** $c > 0$ **and** $n_0 > 0$ **such that** $\forall n \geq n_0 \;\; t(n) \geq cf(n)$**.**

- $t(n) = \Omega(f(n))$ (read "omega") means that the growth rate of $t(n)$ is greater than or equal to the growth rate of $f(n)$.

Def. 3: $t(n) = o(f(n))$ **if** $\lim_{n\to\infty} t(n)/f(n) = 0$.

- $t(n) = o(f(n))$ (read " little-oh ") means that the growth rate of $t(n)$ is strictly smaller than the growth rate of $f(n)$.

Upper bound and Lower bound

- If $t(n) = O(f(n))$ then we say $f(n)$ is an upper bound on $t(n)$.
- If $t(n) = \Omega(f(n))$ then we say $f(n)$ is a lower bound on $t(n)$.

For example, n^2 grows faster than n and thus, n^2 is an upper bound on $n (n = O(n^2))$. Clearly, cn for any constant $c \geq 1$ is also an upper bound on $n (n = O(n))$. We say that cn is a better upper bound on n than n^2 because cn is more close to n. It is important in the analysis of algorithms to find the best upper and lower bounds on the time and space complexities of algorithms.

Examples on $O(f(n))$

- If the steps of operation an algorithm is

$$f(n) = 2n^3 + 10n + 100$$

then

$$t(n) = O(n^3).$$

- The algorithm might consist of modules, $O(n^3)$, $O(n)$, $O(1)$ and it means

$$O(n^3) = O(n^3) + O(n) + O(1).$$

Examples on $O(f(n))$ 2

Given $t_1(n) = O(f(n))$ and $t_2(n) = O(g(n))$,

$$t_1(n) + t_2(n) = \max\{O(f(n)), O(g(n))\}$$

and

$$t_1(n) * t_2(n) = O(f(n) * g(n)).$$

Examples on $O(f(n))$ 3

- If $t(n)$ is a polynomial in n of degree k then

$$t(n) = O(n^k).$$

- For any constant k,

$$\log^k n = O(n).$$

Examples on $O(f(n))$ 4

- Since " Big-Oh " is used to express the growth rate of a function, we should not include constants or low-order terms inside a Big-Oh, such as

$$t(n) = O(2n) \text{ or } t(n) = O(n + \log n).$$

- The correct form for the above is

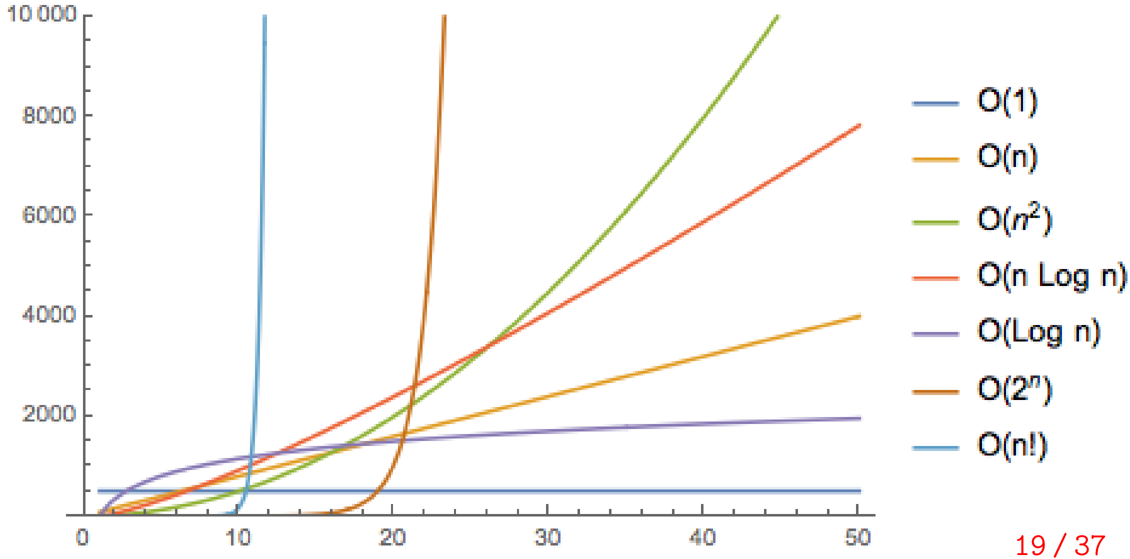
$$t(n) = O(n).$$

- Notice that a constant is denoted by $O(1)$.

Some typical growth rates

Function	Name
c	Constant
$\log n$	Logarithmic (base : 2, e, 10)
n	Linear
$n \log n$	(read: en-log-en)
n^2	Quadratic (read: en-square)
n^3	Cubic
2^n	Exponential (read: two-to the power-en)

Comparisons of different orders of complexity



The sizes of problems that can be solved in one second, one minute, and one hour by five algorithms with different time complexities. The computer speed is 1000 steps/s and $\log$ is based on 2.

Algorithm	Time complexity	Maximum problem size		
		1 sec.	1 min.	1 hour
A_1	n	1000	6×10^4	3.6×10^6
A_2	$n \log n$	140	4893	2.0×10^5
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Table shows the **increase in the size** of the problem we can solve due to the 10-fold increase in speed of computer.

Algorithm	Time complexity	Maximum problem size before speed-up	size after speed-up
A_1	n	s_1	$10s_1$
A_2	$n \log n$	s_2	$\sim 10s_2$ for large s_2
A_3	n^2	s_3	$3.16s_3$
A_4	n^3	s_4	$2.15s_4$
A_5	2^n	s_5	$s_5 + 3.3$

Running Times of Different Classes of Algorithms

Algorithm	Time complexity	# of Operations for $n = 10^6$	Time at 10^6 op/s
A_0	$O(1)$	1	1μs
A_1	$O(n)$	10^6	1 sec.
A_2	$O(n \log n)$	2×10^7	20 sec.
A_3	$O(n^2)$	10^{12}	11.6days
A_4	$O(n^3)$	10^{18}	$32,000$years
A_5	$O(2^n)$	$10^{301,030}$	$10^{301,006}$ times the age of the universe

for/while loop statement.

For the statement for $(i = 1; i <= m; i++)S$, if the computation time of S is $t_i(n)$ for each i then the computation time of the <u>for</u> statement is $\sum_{i=1}^{m} t_i(n)$.
If $t_i(n) = t(n)$ for all i then the computation time of the loop is $mt(n)$.

Consecutive statements.

Let $t_1(n)$ and $t_2(n)$ be the computation times of two consecutive statements, respectively. The total computation time of the two statements is $t_1(n) + t_2(n)$.

if / else statement.

For the statement of if (condition) S_1 else S_2, let $t_1(n)$ and $t_2(n)$ be the computation times of S_1 and S_2, respectively. The computation time of the if statement is $\max\{t_1(n), t_2(n)\}$.

Selection Sort (Bubble Sort)

- Assume n keys are put in an array of n elements.
- We first find the smallest key from the array and exchange it with the key in the first element of the array.
- Next, we find the second smallest key and exchange it with the key in the second element of the array.
- We continue in this way until the entire array is sorted.

Selection Sort

```
selection(int A[], int n){
    int i, j, min, t;
    for (i=1;i<n;i++){
        min=i;
        for (j=i+1; j<=n; j++)
            if (A[j]< A[min])
                min=j;  t=A[min];
                A[min]=A[i]; A[i]=t;
    }
}
```

Time Complexity of Selection Sort Algorithm

The if statement takes constant time. The inner for loop takes $O(n-i)$ time to find the minimum key in $A[i], \ldots, A[n]$. Therefore, the time complexity of selection sort algorithm is

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

More precisely: # of comparisons is
$(n-1) + (n-2) + \cdots + 2 + 1 =?$

Merge Sort (Quicksort)

- Assume two sorted arrays $A[1], \ldots, A[m]$ and $B[1], \ldots, B[n]$ of keys are given.
- To merge the keys in A and B into a third array $C[1], \ldots, C[m+n]$ in which the keys are sorted,
  - we choose the smallest key from A and B and move the key to $C[1]$.
  - Choose the smallest key from the remaining keys in A and B and move the key to $C[2]$.
  - Continue the process of choosing the smallest for C from the remaining keys of A and B until all keys are moved to C.

Merge Sort

The following is a direct implementation of the above merging strategy.

```
i=1;j=1;
A[m+1]=max; B[n+1]=max;
for (k=1; k<=m+n; k++)
    if (A[i] < B[j]) C[k]=A[i++];
    else C[k]=B[j++];
```

We call the above merging an m-by-n merging.
Obviously, an m-by-n merging takes $O(m+n)$ time.

A Recursive Merge Sort Program

```
mergesort(int A[],int left,int right){
    int i,j,k,mid;
    if (right>left) {
    mid=(right+left)/2;
    mergesort(A,left,mid);
    mergesort(A,mid+1,right);
    for (i=left;i<=mid;i++) B[i]=A[i];
        for (i=mid+1,j=right;i<=right;i++,j--)B[i]=A[j];
          i=left;j=right;
        for (k = left; k <= right; k++)
          if (B[i]<B[j]) A[k]=B[i++]; else A[k]=B[j--];
} }
```

Time Complexity of Merge Sort Algorithm

- There are two recursive calls, each of them sorts a sequence of $n/2$, and the statements after the two recursive calls take $O(n)$ time.
- Let $t(n)$ be the time complexity of the algorithm.

$$t(n) = 2t(n/2) + cn$$

and $t(2) = O(1)$, where c is a constant. Solving the equation, $t(n) = O(n \log n)$.

Master Theorem

Let $T(n)$ be a monotonically increasing function that satisfies

$$\begin{aligned} T(n) &= aT(\frac{n}{b}) + f(n) \\ T(1) &= c \end{aligned}$$

where $a \geq 1, b \geq 2, c > 0$. If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

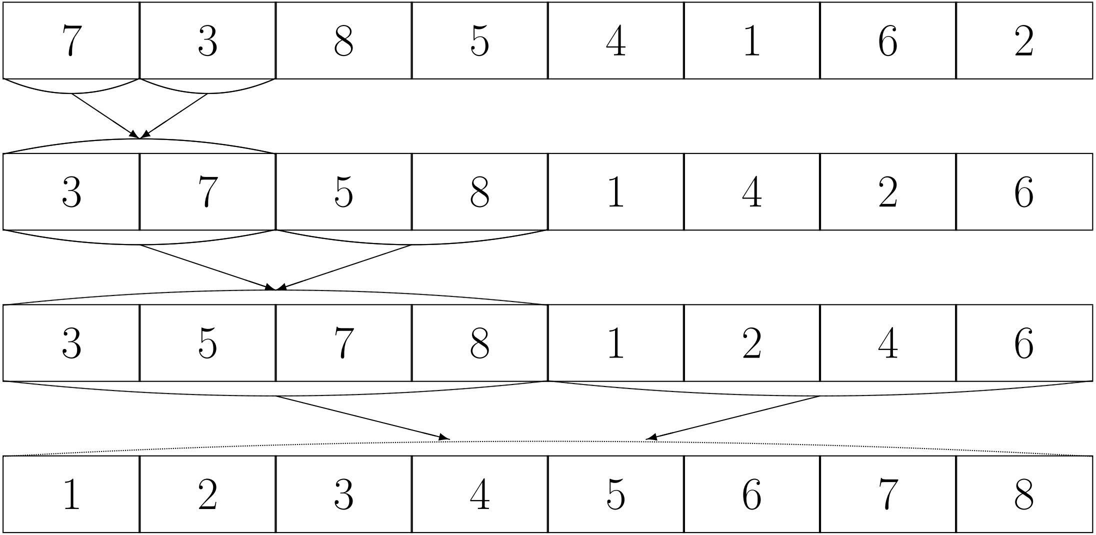
$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

A Non-recursive Merge Sort Algorithm

Assume array $A[1], \ldots, A[n]$ of keys is given.

- we starts from 1-by-1 merging to get $n/2$ sorted subarrays of 2 elements,
- then 2-by-2 merging to get $n/4$ sorted subarrays of 4 elements,
- then 4-by-4 merging to get $n/8$ sorted subarrays of 8 elements, etc., until the whole array is sorted.

A Non-recursive Merge Sort



A Non-recursive Merge Sort Program

```
size=1;
while (size<n) {
   k=0;
   while (k<=(n-2*size)){
      for (i=1; i<=size; i++){
         X[i]=A[k+i];Y[i]=A[k+size+i]; }
      merge();        k=k+2*size;     }
   size=size*2; }
void merge(){
   int p,q;
   X[size+1]=max; Y[size+1]=max; p=1; q=1;
   for (i=1;i<=2*size;i++){
        if (X[p]<Y[q]) A[k+i]=X[p++];
        else A[k+i]=Y[q++]; }
```

Merge Sort Time Complexity

It is easy to check that the time complexity of the non-recursive merge sort is $O(n \log n)$ as well.