Algorithm: What is it?

Example: Linear searching in an array

```
int lsearch(vector<int> A, int key) {

    for (int i = 0; i < A.size(); i++)
       if (A[i] == key) return i;
    return -1;
}
```

A more readable and concise description:

> *Scan the entries of A and compare each with the input key.*

Example: Finding the median score of the class

```
double median(vector<double> S) {
     double temp;

     for (int i = 0; i < S.size()-1; i++)
         for (int j = i+1; j > 0; j--)
             if (S[j] < S[j-1]) {
                temp = S[j-1];
                S[j-1] = S[j];
                S[j] = temp;
               }
             else
                break;
     return S[S.size()/2];
}
```

A more readable and concise description:

> *Sort the scores in the array* `S[0..size-1]` *and then return the entry* `S[size/2]`.

When specifying an algorithm

- An abstraction of what a working program should do.
- A list of high-level steps that you (and the readers) are confident about how to perform.
- Usually in words, sometimes in pseudo-code.

Advantages

- Emphasis on the main logic instead of details.
- A more compact description.
- Modular: fully understood algorithms become steps in complex algorithms.

Why studying algorithms?

Maximum contiguous subsequence sum problem: Given an array $A[0..n-1]$ of n integers (positive/negative/zero), find a contiguous subsequence with the maximum sum.

Algorithm 1: brute-force

For each i, compute $S_{ij} = \sum_{k=i}^{j} A[k]$ for $i \le j \le n-1$. Select max$\{S_{ij}\}$.

We call this the brute-force algorithm because it solves the problem by computing according to the definition. This is the obvious approach and is often the easiest to think of too.

Algorithm 2

For each i, initialize $S_{ii} = A[i]$ and then compute $S_{ij} = S_{i,j-1} + A[j]$ for $i+1 \le j \le n-1$. Select max$\{S_{ij}\}$.

Which algorithm would you prefer based on efficiency concern?

Comparing the two algorithms

Algorithm1: Number of additions required:

$$
\begin{aligned}
& \textstyle\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j-i) \\
= \;& \textstyle\sum_{i=0}^{n-1} \frac{(n-i-1)(n-i)}{2} \\
= \;& \textstyle\sum_{i=0}^{n-1} \frac{(n^2-n)-(2n-1)i+i^2}{2} \\
= \;& \tfrac{1}{2}n(n^2-n) - \tfrac{1}{4}n(n-1)(2n-1) + \\
& \tfrac{1}{12}n(n-1)(2n-1) \\
= \;& \tfrac{1}{6}n^3 - \tfrac{1}{6}n
\end{aligned}
$$

Algorithm 2: Number of additions required:

$$
\begin{aligned}
& \textstyle\sum_{i=0}^{n-2} (n-i-1) \\
= \;& \tfrac{1}{2}n^2 - \tfrac{1}{2}n
\end{aligned}
$$

The following tables illustrate the difference in efficiency.

Efficiency in the number of additions

n	Alg 1	Alg 2	Ratio
10	165	45	3.67
100	166650	4950	33.67
200	1333300	19900	67
500	20833250	124750	167
1000	166666500	499500	333.67
2000	1333333000	1999000	667

Efficiency in seconds

n	Alg 1	Alg 2	Ratio
10	0.00	0.01	–
100	0.02	0.01	2
200	0.13	0.00	–
500	1.92	0.03	64
1000	15.33	0.09	170.3
2000	121.80	0.35	348

Time Complexity Analysis of Algorithms

Motivation

Want to identify the faster solution to a problem.

What is the efficiency of a program?

It should give a prediction how fast it can run given an input.

Different approaches

- *Empirical*: Run an implemented system on real-world data. Notion of benchmarks.
- *Simulational*: Run an implemented system on simulated data. For example, testing of your programming assignments.
- *Analytical*: Use theoretic-model data with a theoretical-model system. We will focus on this.

Considerations...

- The size of the input affects the running time.
- The content of the input affects the running time.
- Different machines have different speed.
- Programmer skill affects the running time.

Problem, Instances, and Input size

Problem

Like a class in C++, a problem refers to an abstract description of a task to be tackled. For example, sorting is an abstract problem.

Instance of a problem

Like an object in C++ which is an instance of a class, an instance of a problem is defined when a specific input is given. For example, sorting `10, 2, 3, 7, 4` is an instance of the sorting problem.

Input size

The input size of a problem refers to the number of items in the problem input. For example, for the sorting problem, it refers to the number of items to be sorted. As another example, the input size of the problem of multiplying two matrices refers to the total number of entries in the two input matrices.

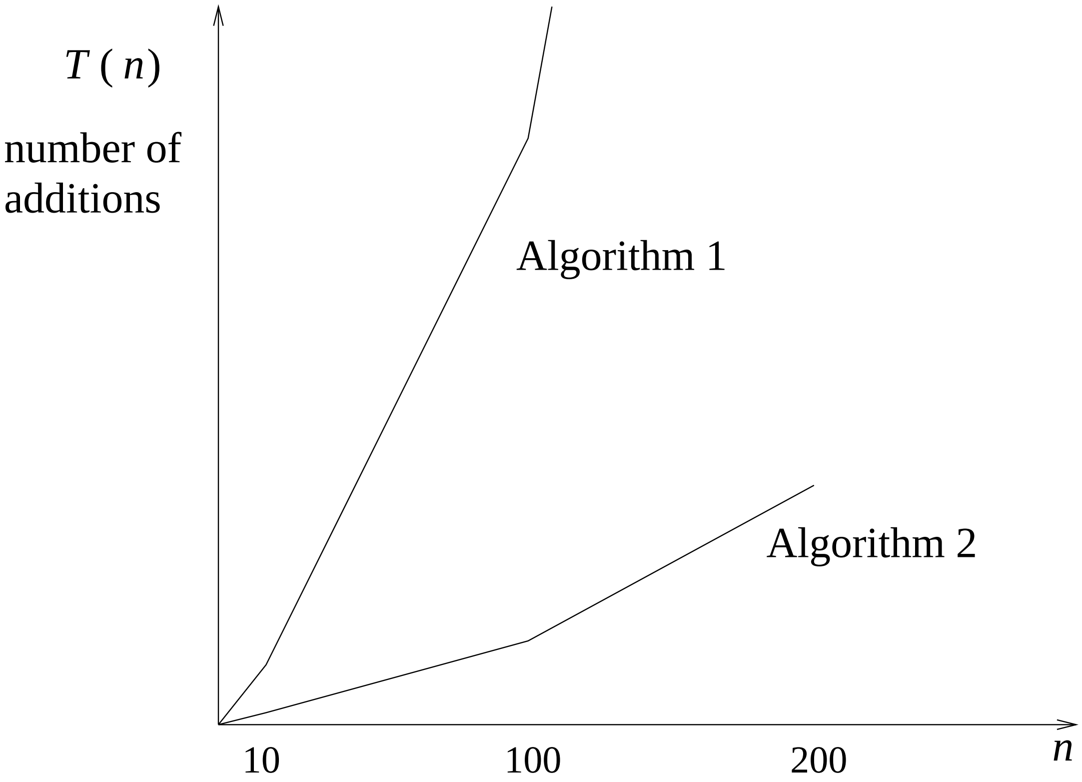
Analytical Time Complexity Analysis

- We would like to compare efficiencies of different algorithms for the same problem, instead of different programs or implementations. This removes dependency on machines and programming skill.

- It becomes meaningless to measure absolute time since we do not have a particular machine in mind. Instead, we measure the number of steps. We call this the *time complexity* or *running time* and denote it by $T(n)$.

- We would like to estimate how $T(n)$ varies with the input size n.

A plot of the rough number of additions of Algorithm 1 and Algorithm 2 for the maximum contiguous subsequence sum problem against the input size.



Algorithm 2 is clearly superior. The reason is that $T(n)$ for Algorithm 1 is roughly proportional to n^3 while $T(n)$ for Algorithm 2 is roughly proportional to n^2. Since n^3 grows much more rapidly, we expect Algorithm 1 to take much more time than Algorithm 2 when n increases.

Asymptotic

If A is a much better algorithm than B, then it is not necessary to calculate $T_A(n)$ and $T_B(n)$ exactly. As n increases, since $T_A(n)$ will grow much more rapidly, $T_B(n)$ will always be less than $T_A(n)$ for large enough n.

Thus, it suffices to measure the *growth rate* of time complexity to get a rough comparison.

Big-Oh

$f(n) = O(g(n))$:

> There exists constant $c > 0$ and n_0 such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$.

Algorithm 1 runs in $O(n^3)$ time. Algorithm 2 runs in $O(n^2)$ time.

When estimating the growth rate of $T(n)$ using big-Oh:

- Ignore the low order terms.
- Ignore the constant coefficient of the most significant term.
- The remaining term is the estimate.

For example, Algorithm 1 runs in $O(n^3)$ time and Algorithm 2 runs in $O(n^2)$ time. More examples:

- $n^2/2 - 3n = O(n^2)$
- $1 + 4n = O(n)$
- $\log_{10} n = \frac{\log_2 n}{\log_2 10} = O(\log_2 n) = O(\log n)$
- $\sin n = O(1)$, $10 = O(1)$, $10^{10} = O(1)$.
- $\sum_{i=1}^{n} i^2 \leq n \cdot n^2 = O(n^3)$
- $\sum_{i=1}^{n} i \leq n \cdot n = O(n^2)$
- 2^{10n} is not $O(2^n)$
- $7n^2 + 10n + 3 = O(n^2) = O(n^3) = O(n^4)$

$f(n) = \Omega(g(n))$ (big-Omega):

There exists constant $c > 0$ and n_0 such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$.

$f(n) = \Theta(g(n))$ (big-Theta):

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

For example, Algorithm 1 runs in $\Omega(n^3)$ time and Algorithm 2 runs in $\Omega(n^2)$ time. Thus, Algorithm 1 runs in $\Theta(n^3)$ and Algorithm 2 runs in $\Theta(n^2)$.

Binary Search vs Linear Search

```
int bsearch(vector<int> A, int key) {
    int low = 0, high = A.size() - 1;

    while (low <= high) {
       int mid = (low + high)/2;
       if (A[mid] < x)
           low = mid + 1;
       else if (x < A[mid])
           high = mid - 1;
       else
           return mid;
    }
    return -1;
}
```

Binary search is a faster procedure to find an item in a sorted vector than linear search. How much faster?

Efficiency in seconds per 100000 random trials

n	Linear	Binary	Ratio
100	1.77	0.27	6.6
500	8.03	0.3	26.7
1000	14.53	0.32	45.4
2000	23.9	0.35	68.3
5000	28.07	0.41	68.46
10000	28.24	0.44	64

Analytic Comparison

```
int lsearch(vector<int> A, int key) {

    for (int i = 0; i < A.size(); i++)
       if (A[i] == key) return i;
    return -1;
}
```

The problem with analyzing **`lsearch`** is that the number of steps taken depends on both the input size and the input content.

For each input size n, consider the *worst-case input* that gives the largest time complexity $T(n)$:

- gives an upper bound on the running time.
- usually easier to analyze.

$T(n)$ is then called the *worst-case time complexity*.

The time complexity of **`lsearch`** is $O(n)$.

Although we exploit the concept of worst-case input, we do not need to exhibit a worst-case input to perform a worst-case analysis.

- `bsearch` takes the most time if the while-loop iterates until the while-condition fails.
- Each iteration of the loop executes at most a constant number of instructions and each instruction can be done in $O(1)$ time..
- So the time complexity of `bsearch` is $O(1)$ times the maximum number of iterations.

It suffices to give an upper bound on the maximum number of iterations of the while-loop:

- Initially, `high - low = n-1`.
- Let N_i and N_{i+1} be the values of `high - low` before the ith and the $(i+1)$th iterations respectively.

  Since either `high = mid-1` or `low = mid+1`, we have

  $$N_{i+1} \leq N_i/2.$$

- This implies that $N_i \leq (n-1)/2^{i-1}$.
- If `high - low` drops to 1 or below, the loop will iterate at most two more times.
- The number of iterations is at most $1+k$, where k is the smallest integer such that $(n-1)/2^{k-1} \leq 1$. So the number of iterations is at most $2+\log_2(n-1)$.

The time complexity of `bsearch` is $O(\log n)$.

Lower Bound for `bsearch`

We know that the time complexity of `bsearch` is $O(\log n)$. How good is this upper bound?

Consider searching for `0` in the sequence

```
0 2 3 ....  n-1
```

- In the first iteration,

$$\texttt{A[mid]} = \lfloor (n-1)/2 \rfloor \geq n/2 - 1.$$

- In the second iteration,

$$\texttt{A[mid]} \geq \lfloor ((n/2) - 2)/2 \rfloor \geq n/4 - 2.$$

- In the ith iteration,

$$\texttt{A[mid]} \geq n/2^i - i.$$

- This implies that

$$\texttt{A[mid] > 0} \text{ if } i \leq \frac{1}{2} \log_2 n.$$

- So there are at least $\frac{1}{2}\log_2 n$ iterations.

In summary, we have shown an input of size n in which `bsearch` must take at least $\frac{1}{2}\log_2 n$ steps.

The time complexity of `bsearch` is $\Omega(\log n)$ and hence $\Theta(\log n)$.