Object-Oriented Programming and Data Structures

COMP2012: Algorithm Analysis

Prof. Brian Mak
Prof. C. K. Tang

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

What Is an Algorithm?

- A clearly specified set of instructions to be followed to solve a problem.
- It takes some input(s) and produces some output(s).
- It may be specified
  - in English;
  - in some programming language as a computer program;
  - in some pseudo codes.
- Program = data structures + algorithms

- Writing a working program is not good enough; it has to be efficient!
- The efficiency of a program may not be an issue when the size of input is small.
- In computer science, we are interested in the asymptotic running time of an algorithm when it is run on a large data set.
- Algorithm analysis allows us
  - to eliminate bad algorithms
  - to pinpoint the bottlenecks, which require more attention during implementation

Selecting the kth Largest Element: Solution 1

Problem: Given a list of N numbers, determine the kth largest number, where $k \leq N$.

Solution 1

STEP 1 : Read N numbers into an array

STEP 2 : Sort the array in descending order

STEP 3 : Return the element in position k

Problem: Given a list of N numbers, determine the kth largest number, where $k \leq N$.

Solution 2

STEP 1 : Read the first k numbers into an array

STEP 2 : Sort the array of size k in descending order

STEP 3 : Read each remaining number one by one into the array,

- if it is smaller than the kth element, then it is ignored
- otherwise, it is placed in its correct position in the array, bumping the last element out of the array.

STEP 4 : Return the element in position k.

Selecting the kth Largest Element: Which Solution?

- Which algorithm is better when
  - $N = 100$ and $k = 100$?
  - $N = 100$ and $k = 1$?
- What happens when
  - $N = 1,000,000$ and $k = 500,000$?

Algorithm Analysis

- We only analyze correct algorithms.
- An algorithm is correct if, for every input instance, it halts with the correct output.
- Incorrect algorithms
  - might not halt at all on some input instances;
  - might halt with other than the desired answer.
- Analyzing an algorithm allows us to predict the resources that it requires.
- Resources include
  - memory
  - communication bandwidth
  - computational time (usually most important)

Running Time Factors

- Factors affecting the running time of an algorithm:
  - computer
  - compiler
  - the specific algorithm used
  - input(s) to the algorithm
- While the content of the input will affect the running time of an algorithm, typically, it is the input size (number of items in the input) that is the main consideration.
  - in sorting: the number of items to be sorted.
  - in matrix multiplication: the total number of elements in the two matrices.

Analysis Model and Approaches

- We assume the following machine model for algorithm analysis:
  - instructions are executed one after another, with no concurrent operations — non-parallel computers.

- 3 analysis approaches
  1. Empirical: run an implemented system on real-world data. (Also used for developing benchmarks).
  2. Simulative: run an implemented system on simulated data.
  3. Analytical: use theoretic-model data with a theoretical model.

Example 1: Running Time Analysis

```
1:  int sum(int N)
2:  {
3:      int partial_sum;
4:      partial_sum = 0;
5:      for (int i = 1; i <= N; i++)
6:          partial_sum += i*i*i;
7:      return partial_sum;
8:  }
```

- Lines 4 and 7: each counts for one time unit.
- Line 6: executed for N times, each time for 4 time units.
- Line 5: a total of $2N + 2$ time units:
  - initialization for 1 unit
  - comparison test for a total of $N + 1$ units
  - increments for a total of N units
- Total cost $= 6N + 4$

Worst- / Average- / Best-Case Analysis

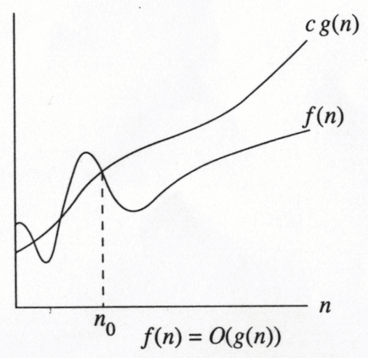
- Worst-case running time of an algorithm
  - the longest running time for any input of size n.
  - an upper bound on the running time for any input; it guarantees that the algorithm will never take longer.
  - example: Sort a set of numbers in increasing order and the input numbers data are in decreasing order.
  - the worst case can occur fairly often, e.g., in searching a database for a particular piece of information.
- Best-case running time of an algorithm
  - example: sort a set of numbers in increasing order, and the input numbers are already in increasing order.
- Average-case running time of an algorithm
  - may be difficult to define what "average" means.

Running-time of Algorithms

- Bounds are computed for algorithms, rather than programs.
- Programs are just implementations of algorithms, and almost always the details of a program do not affect the bounds.
- For analysis purpose, algorithms are often written in pseudo-codes; we'll use something almost like C++.
- Bounds are computed for algorithms, rather than problems.
- A problem can probably be solved with several algorithms, and some are more efficient than others.
- 3 types of bounds:
  - Upper bound: $O(g(N))$
  - Lower bound: $\Omega(g(N))$
  - Tight bound: $\Theta(g(N))$

- The idea is to establish a relative order among functions for large value of input size N.
- $f(N)$ grows no faster than $g(N)$ for large N if

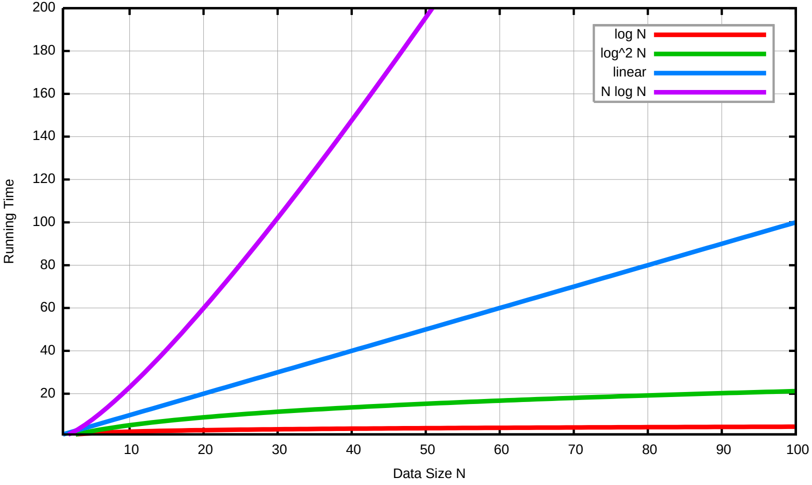
$$\exists c, n_0 > 0 \quad \text{such that} \quad f(N) \leq c\, g(N) \quad \text{when} \quad N \geq n_0$$



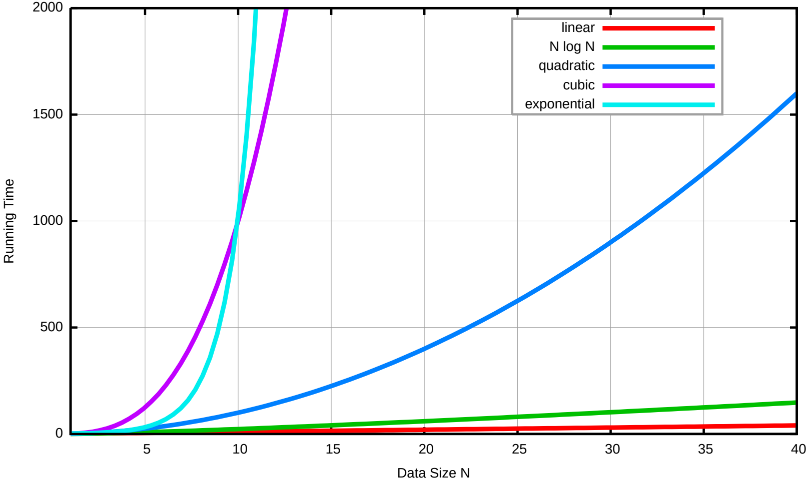
Typical Growth Rates of Algorithm Running Time

Function	Name
c	constant
$\log N$	logarithmic
$\log^2 N$	log-squared
N	linear
$N \log N$	
N^2	quadratic
N^3	cubic
2^N	exponential

Typical Growth Rates of Algorithm Running Time ..



Typical Growth Rates of Algorithm Running Time ...



Asymptotic Upper Bound: Big-Oh

- If $f(N) = O(g(N))$, then there are positive constants c and n_0 such that

$$f(N) \leq c\, g(N) \quad \text{when} \quad N \geq n_0$$

- The growth rate of $f(N)$ is less than or equal to the growth rate of $g(N)$.
- $g(N)$ is an upper bound on $f(N)$.
- E.g., if $f(N) = O(N^2)$, we say that the order of $f(N)$ is "N-squared" or "big-oh N-squared".

Some Rules for Big-Oh

- Ignore the lower order terms.
- Ignore the coefficients of the highest-order term.
- No need to specify the base of logarithm: changing the base of a logarithm changes the value of the logarithm by only a constant factor.
- Example: If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$,

$$\begin{aligned} T_1(N) + T_2(N) &= max(O(f(N)), O(g(N))) \\ \text{and} \quad T_1(N) \times T_2(N) &= O(f(N) \times g(N)) \end{aligned}$$

Part I

Examples

"I used to lead by example but it was too much work."

Example 2.1: Linear Search

```
/* File: linear-search.cpp
 * Given a sorted array, data, of size N in non-descending order,
 * find the value x by searching the array elements sequentially
 */

const int NOT_FOUND = -1;

int linear_search(const int data[ ], int N, int x)
{
    for (int i = 0; i < N; i++)                              // O(N)
    {
        if (data[i] == x)
            return i;
    }

    return NOT_FOUND;
}
```

Example 2.2: Iterative Binary Search

```
/* File: iterative-bsearch.cpp
 * Given a sorted array, data, of size N in non-descending order,
 * find the value x by binary search iteratively.
 */
const int NOT_FOUND = -1;
int bsearch(const int data[ ], int N, int x)
{
    int first = 0, last = N - 1;
    while (first <= last)           // No. of iterations = ceiling[log(N-1)]
    {                                   // O(1) operations inside the loop
        int mid = (first + last)/2;
        if (data[mid] == x)
            return mid;                                    // Value found!
        else if (data[mid] > x)
            last = mid - 1;         // Set up for searching the lower half
        else
            first = mid + 1;        // Set up for searching the upper half
    }
    return NOT_FOUND;
}
```

Example 2.3: Recursive Binary Search

```
/* File: recursive-bsearch.cpp
 * Given a sorted array, data, of size N in non-descending order,
 * find the value x by binary search recursively.
 */

const int NOT_FOUND = -1;
int bsearch(const int data[ ], int first, int last, int x)
{
    if (last < first)
        return NOT_FOUND;                  // Base case #1: O(1)

    int mid = (first + last)/2;

    if (data[mid] == value)
        return mid;                        // Base case #2: O(1)
    else if (data[mid] > value)            // Search the lower half: T(N/2)
        return bsearch(data, first, mid-1, value);
    else                                   // Search the upper half: T(N/2)
        return bsearch(data, mid+1, last, value);
}
```

Example 2.3: Recursive Binary Search Analysis

- Let $T(N)$ = running time of recursive bsearch over a list of N items. Therefore,
- Recurrence equation: $T(N) = T(N/2) + 1$
- Ending condition: $T(1) = 1$
- Therefore,

$$\begin{aligned} T(N) &= T(N/2) + 1 \\ &= T(N/4) + 1 + 1 \\ &= T(N/8) + 1 + 1 + 1 \\ &\vdots \\ &= T(N/2^k) + k \end{aligned}$$

  where k is the number of recursions.
- Asymptotically, $N/2^k = 1 \Rightarrow k = \log N$.
- That is, $T(N) = T(N/2^k) + k = T(1) + \log N = 1 + \log N$.
- Since each recursion performs $O(1)$ operations, therefore the running time of the whole algorithm is $O(\log N)$.

Example 3: Maximum Sub-sequence Sum (MSSS)

- Problem: Given a sequence of (+ve and -ve) integers: $\{A_1, A_2, \ldots, A_n\}$, find the maximum sum among all the sub-sequences. That is,

$$\sum_{k=i}^{j} A_k$$

- For convenience, the maximum sub-sequence sum is considered 0 if it is negative.
- E.g., for the input $\{-2, 11, -4, 13, -5, -2\}$, the answer is 20 given by the sub-sequence A_2 through A_4.

```
#include <iostream>                    /* File: max-subseq-sum-main.cpp */
using namespace std;

int max_subseq_sum_cubic(const int data[ ], int N);
int max_subseq_sum_quadratic(const int data[ ], int N);
int max_subseq_sum_nlogn(const int data[ ], int left, int right);
int max_subseq_sum_linear(const int data[ ], int N);

int data[ ] = { 4, -3, 5, -2, -1, 2, 6, -2 };

int main( )
{
    const int N = sizeof(data)/sizeof(int);

    cout << "cubic:   " << max_subseq_sum_cubic(data, N) << endl;
    cout << "quadratic:   " << max_subseq_sum_quadratic(data, N) << endl;
    cout << "n log(n):   " << max_subseq_sum_nlogn(data, 0, N-1) << endl;
    cout << "linear:   " << max_subseq_sum_linear(data, N) << endl;
    return 0;
}
```

Example 3: Max Sub-sequence Sum — Sample Run

Input Size	Algorithm Time 1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 100$	0.000159	0.000006	0.000005	0.000002
$N = 1,000$	0.095857	0.000371	0.000060	0.000022
$N = 10,000$	86.67	0.033322	0.000619	0.000222
$N = 100,000$	NA	3.33	0.006700	0.002205
$N = 1,000,000$	NA	NA	0.074870	0.022711

MSSS Algorithm 1: Brute-force Method — $O(N^3)$

```
/* File: max-subseq-sum-cubic.cpp */
// Exhaustively evaluate all possible sub-sequences
int max_subseq_sum_cubic(const int data[ ], int N)
{
    int max_sum = 0;

    for (int i = 0; i < N; ++i)             // i = start of a sub-sequence
        for (int j = i; j < N; ++j)         // j = end of a sub-sequence
        {
            int sum = 0;                    // Sum of data[i] to data[j]
            for (int k = i; k <= j; ++k)
                sum += data[k];             // => O(N^3)

            if (sum > max_sum)
                max_sum = sum;
        }

    return max_sum;
}
```

MSSS Algorithm 2: Improved Brute-force — $O(N^2)$

```
/* File: max-subseq-sum-quadratic.cpp */
int max_subseq_sum_quadratic(const int data[ ], int N)
{
    int max_sum = 0;

    for (int i = 0; i < N; ++i)               // i = start of a sub-sequence
    {
        int sum = 0; // Of the longest subsequence starting from data[i]

        for (int j = i; j < N; ++j)           // j = end of a sub-sequence
        {
            sum += data[j];                   // ⇒ O(N^2)

            if (sum > max_sum)
                max_sum = sum;
        }
    }

    return max_sum;
}
```

MSSS Algorithm 3: Divide-and-Conquer Method

- Split the problem into two roughly equal sub-problems, which are then solved recursively.
- Patch together the 2 solutions of the 2 sub-problems with some extra work to arrive at a solution for the whole problem.
- In this case, the extra work is to also consider the maximum sum among all the sub-sequences that run across the border between the 2 halves.

First Half	Half
4 −3 5 −2	−1 2 6 −2

- At the end, the overall MSSS (11) is the maximum among
  - MSSS computed entirely from the left half (6),
  - MSSS computed entirely from the right half (8), and
  - MSSS computed across the border of the 2 halves (11).

MSSS Algorithm 3: Divide-and-Conquer — $O(N \log N)$

```
/* File: max-subseq-sum-nlogn.cpp */
inline int max(int a, int b) { return (a > b) ? a : b; }

int max_subseq_sum_nlogn(const int data[ ], int left, int right)
{
    if (left == right)                                    // Base case
        return (data[left] > 0) ? data[left] : 0;

    // Recursion on the left and right subsequences
    int center = (left + right)/2;
    int left_max_sum = max_subseq_sum_nlogn(data, left, center);
    int right_max_sum = max_subseq_sum_nlogn(data, center+1, right);

    // Find the max subsequence across the border
    int max_border_sum =
        find_left_max_border_sum(data, center, left) +
        find_right_max_border_sum(data, center, right);

    return max( max(left_max_sum, right_max_sum), max_border_sum);
}
```

```
/* File: max-subseq-sum-nlogn-border.cpp */
int find_left_max_border_sum(const int data[ ], int center, int left)
{
    int left_max_border_sum = 0, left_border_sum = 0;
    for (int i = center; i >= left; --i)                    // ⇒ O(N)
    {
        left_border_sum += data[i];
        if (left_border_sum > left_max_border_sum)
            left_max_border_sum = left_border_sum;
    }
    return left_max_border_sum;
}

int find_right_max_border_sum(const int data[ ], int center, int right)
{
    int right_max_border_sum = 0, right_border_sum = 0;
    for (int j = center+1; j <= right; ++j)                 // ⇒ O(N)
    {
        right_border_sum += data[j];
        if (right_border_sum > right_max_border_sum)
            right_max_border_sum = right_border_sum;
    }
    return right_max_border_sum;
}
```

MSSS Algorithm 3: Divide-and-Conquer Method Analysis

- Recurrence equation: $T(N) = 2T(N/2) + N$ as the patch work is $O(N)$
- Ending condition: $T(1) = 1$
- Thus, we have

$$\begin{aligned} T(N) &= 2T(N/2) + N \\ &= 4T(N/4) + 2N \\ &= 8T(N/8) + 3N \\ &\vdots \\ &= 2^k T(N/2^k) + kN \end{aligned}$$

  where k is the number of splits.
- Asymptotically, $N/2^k = 1 \Rightarrow k = \log N$.
- $T(N) = 2^k T(N/2^k) + kN = NT(1) + N \log N = N + N \log N$.
- Hence, the running time of the whole algorithm is $O(N \log N)$.

MSSS Algorithm 4: Simply Smart — $O(N)$

```
/* File: max-subseq-sum-linear.cpp */
int max_subseq_sum_linear(const int data[ ], int N)
{
    int max_sum = 0;
    int sum = 0;                              // Sum of data[i] to data[j]

    for (int j = 0; j < N; ++j)               // j = end of a sub-sequence
    {
        sum += data[j];  // Of the longest subsequence ending at data[j]
                                                             // ⇒ O(N)

        if (sum > max_sum)
            max_sum = sum;
        else if (sum < 0)      // Discard prefix subsequence with -ve sum
            sum = 0;
    }

    return max_sum;
}
```