COMP 3711 Design and Analysis of Algorithms

Introduction

What is an Algorithm?

Definition:

An algorithm is a recipe for doing something.

An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions.

Comes from al-Khwārizmi, the name of a 9th century Persian mathematician, astronomer, geographer, and scholar

Algorithm for Adding Two Numbers

Input: Two numbers x and y (potentially very long), each consisting of n digits: $x = \overline{x_n x_{n-1} \dots x_1}, y = \overline{y_n y_{n-1} \dots y_1}$

Output: A number $z = \overline{z_{n+1} z_n \dots z_1}$, such that $z = x + y$.

```
c ← 0
for i ← 1 to n  //i is an index on the
decimal digits, starting from the right
    z_i ← x_i + y_i + c
    if z_i ≥ 10 then c ← 1, z_i ← z_i − 10
                else c ← 0
z_{n+1} ← c
```

```
  529501233
+ 612345678
-----------
 1141846911
```

You've been running algorithms all your life!

The Sorting Problem

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

[4 8 2 7 5 6 9 3]

Output: Array $A[1 \ldots n]$ of elements in sorted order (ascending)

[2 3 4 5 6 7 8 9]

We always use ascending order, but all algorithms work for descending order with minor modifications

Sorting is a very important (sub)routine that comes up all the time.

- One of the early **Killer-Apps** of Computing
  - In the 1960s', computer manufacturers estimated that more than 25% of their machines' running times were spent on sorting
- Even now, sorting is a common application AND a standard subroutine used in solutions to many other problems
- The Sorting Problem is also a good testbed to test out algorithmic techniques

Selection Sort

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

Output: Array $A[1 \ldots n]$ of elements in sorted order (ascending)

(*) n-1 passes
In pass i, find smallest item in $A[i \ldots n]$ and swap it with $A[i]$

```
Selection-Sort(A):
for i ← 1 to n − 1
    for j ← i + 1 to n
        if A[i] > A[j] then
            swap A[i] and A[j]
```

Implementation

1st Pass: (5 2 8 6 7 1) → (2 5 8 6 7 1) → (2 5 8 6 7 1) → (2 5 8 6 7 1) → (2 5 8 6 7 1) → (1 5 8 6 7 2)

2nd Pass: (1 5 8 6 7 2) → (1 5 8 6 7 2) → (1 5 8 6 7 2) → (1 5 8 6 7 2) → (1 2 8 6 7 5)

3rd Pass: (1 2 8 6 7 5) → (1 2 6 8 7 5) → (1 2 6 8 7 5) → (1 2 5 8 7 6)

4th Pass: (1 2 5 8 7 6) → (1 2 5 7 8 6) → (1 2 5 6 8 7)

5th Pass: (1 2 5 6 8 7) → (1 2 5 6 7 8) → (1 2 5 6 7 8)

Section Sort (in ascending order)

Step 1: Find the minimum in array $A[1 \dots n]$ and swap $A[1]$ and $A[pos]$

Cost of Step 1: n-1 comparisons

Step 2: Find the minimum in array $A[2 \dots n]$ and swap $A[2]$ and $A[pos]$

Cost of Step 2: n-2 comparisons

Step *i*: Find the minimum in array $A[i \dots n]$ and swap $A[i]$ and $A[pos]$

Cost of Step *i*: n-*i* comparisons

Observation: after step *i*, $A[1 \dots i]$ is sorted

Step n-1: Find the minimum in array $A[n-1 \dots n]$ and swap $A[n-1]$ and $A[pos]$

Cost of Step n-1: 1 comparison

Total cost of the algorithm=Total number of comparisons=(n-1)+(n-2)+…+2+1

This is the arithmetic series.

Q: Is the cost (number of comparisons) always the same for any array of size n?

Correctness of Selection Sort

Claim: When Selection sort terminates, the array is sorted.

Proof: By induction on n.

When n = 1 the algorithm is obviously correct

Assume that the algorithm sorts every array of size n-1 correctly

Now consider what the algorithm does on $A[1 \ldots n]$

1. It firsts puts the smallest item in $A[1]$

2. It then runs Selection sort on $A[2 \ldots n]$
   - By induction, this sorts the items in $A[2 \ldots n]$

3. Since $A[1]$ is smaller than every item in $A[2 \ldots n]$, all the items in $A[1 \ldots n]$ are now sorted.

Running Time of Selection Sort

What is meant by running "time"?

Seconds?

Total Operations Performed?

Memory Accesses?

```
Selection-Sort(A):
1. for i ← 1 to n − 1
2.     for j ← i + 1 to n
3.         if A[i] > A[j] then
4.             swap A[i] and A[j]
```

We will count number of comparisons, i.e., # of times $A[i] > A[j]$ is run.
This "dominates" all of the other operations.

For any fixed i, line 3 calls $n - i$ values of j. Total # comparisons is

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

Alternatively, note that line 3 is run exactly once for every possible (i,j) pair with $1 \le i < j \le n$. This is exactly # of ways to choose a pair out of n items which is

$$\binom{n}{2} = \frac{\boldsymbol{n(n-1)}}{\boldsymbol{2}}$$

Insertion Sort

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

Output: Array $A[1 \ldots n]$ of elements in sorted order (ascending)

Insertion is performed by successively swapping $A[i]$ with items to its left until an item smaller than $A[i]$ is found.

```
Insertion-Sort(A):
for i ← 2 to n do
    j ← i − 1
    while j ≥ 1 and A[j] > A[j + 1] do
        swap A[j] and A[j + 1]
        j ← j − 1
```

i=2: (5 1 8 6 3 2) → (1 5 8 6 3 2) → (1 5 8 6 3 2)

i=3: (1 5 8 6 3 2) → (1 5 8 6 3 2)

i=4: (1 5 8 6 3 2) → (1 5 6 8 3 2) → (1 5 6 8 3 2)

i=5: (1 5 6 8 3 2) → (1 5 6 3 8 2) → (1 5 3 6 8 2) → (1 3 5 6 8 2) → (1 3 5 6 8 2)

i=6: (1 3 5 6 8 2) → (1 3 5 6 2 8) → (1 3 5 2 6 8) → (1 3 2 5 6 8) → (1 2 3 5 6 8)
→ (1 2 3 5 6 8)

Insertion Sort

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

Output: Array $A[1 \ldots n]$ of elements in sorted order (ascending)

Insertion is performed by successively swapping $A[i]$ with items to its left until an item smaller than $A[i]$ is found.

```
Insertion-Sort(A):
for i ← 2 to n do
    j ← i − 1
    while j ≥ 1 and A[j] > A[j + 1] do
        swap A[j] and A[j + 1]
        j ← j − 1
```

Correctness: After step i, items in A[1..i] are in proper order.
i 'th iteration puts key=A[i] in proper place.

Running Time of Insertion Sort

```
Insertion-Sort(A):
1. for i ← 2 to n do
2.     j ← i − 1
3.     while j ≥ 1 and A[j] > A[j + 1] do
4.         swap A[j] and A[j + 1]
5.         j ← j − 1
```

Number of comparisons at Line 3 depends on the array. It is at most

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

Q: When does the worst case (largest number of comparisons) happen?

If original array is sorted, e.g., (1 2 3 4 5 6), then no item is ever moved.
Each item is only compared to its left neighbor, so line 3 is only run $\boldsymbol{(n-1)}$ times.

Unlike Selection Sort which always uses $\frac{\boldsymbol{n(n-1)}}{\boldsymbol{2}}$ comparisons for each array of size n, the number of comparisons (running time) of Insertion Sort depends on the input array, and ranges between $\boldsymbol{(n-1)}$ and $\frac{\boldsymbol{n(n-1)}}{\boldsymbol{2}}$.

Wild-Guess Sort

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

Output: Array $A[1 \ldots n]$ of elements in sorted order (ascending)

```
Wild-Guess-Sort(A):
π ← [4, 7, 1, 3, 8, 11, 5, ...] //create random permutation
check if A[π[i]] ≤ A[π[i + 1]] for all i = 1, 2, ..., n − 1
if yes, output A according to π and terminate
else Insertion-Sort(A)
```

Q: Can Wild-Guess Sort be faster than insertion sort?

A: Yes, if the guess happens to be correct (highly unlikely).

A: Otherwise, it is slower.

How to evaluate an algorithm / compare two algorithms?

What to measure?

- **Memory** (space complexity)
  - total space
  - working space (excluding the space for holding inputs)
- **Running time** (time complexity)

How to measure?

- **Empirical** – depends on actual implementation, hardware, etc.
- **Analytical** – **depends only on the algorithms, focus of this course**

Comparing two algorithms is not (usually) a simple matter!

- Even for same input size n, different inputs can lead to different running times
- Calculating exact # of instructions executed is very difficult/tedious

Best-Case Analysis

Best case: An instance for a given size n that results in the fastest possible running time.

Example (insertion sort): Input already sorted

```
Insertion-Sort(A):
for i ← 2 to n do
    j ← i − 1
    while j ≥ 1 and A[j] > A[j + 1] do
        swap A[j] and A[j + 1]
        j ← j − 1
```

"key" is only compared to element to its immediate left, so

$$T(n) = n - 1 = \Theta(n).$$

Q: What is the best case running time of Wild-Guess Sort?

Worst-Case Analysis

Worst case: An instance for a given size n that results in the slowest possible running time.

Example (insertion sort): Input inversely sorted

```
Insertion-Sort(A):
for i ← 2 to n do
    j ← i − 1
    while j ≥ 1 and A[j] > A[j + 1] do
        swap A[j] and A[j + 1]
        j ← j − 1
```

"key" is compared to every element preceding it, so

$$T(n) = \Theta\left(\sum_{i=2}^{n}(i-1)\right) = \Theta\left(\frac{n(n-1)}{2}\right) = \Theta(n^2).$$

Average-Case Analysis

Average case: Running time averaged over all possible instances for the given size, assuming some probability distribution on the instances.

Example (insertion sort): assuming that each of the $n!$ permutations of the n numbers is equally likely

Rigorous analysis is complicated, but intuitively, "key" is compared, on average to half the items preceding it, so

$$T(n) = \Theta\left(\sum_{i=2}^{n} \frac{i-1}{2}\right) = \Theta\left(\frac{n(n-1)}{4}\right) = \Theta(n^2).$$

Three Kinds of Analyses

Best case: Clearly useless

Worst case: Commonly used

Gives running time guarantee ***independent of actual input***

- Fair comparison among different algorithms
- Not perfect: For some problems, worst-case input never occurs in real life; some algorithms with bad worst-case running time actually work well in practice (e.g. the simplex algorithm for linear programming)
- **Worst-case analysis is the default and the one used in this class. From now on when we refer to the running time of an algorithm, we imply the worst case running time.**

Average case: Sometimes used

- Needs to assume some input distribution:
  real-world inputs are seldom uniformly random!
- Analysis is complicated
- Will see examples later

More on Worst-Case Analysis (Expanded)

What does each of these statements mean? (*n is the input/problem size)*

The algorithm's worst case running time is $O(f(n))$

- On all inputs of (large) size n, the running time of the algorithm is $\leq c \cdot f(n)$

Further expanded:
There exist constants $c > 0, n_0 \geq 0$, *such that for any* $n \geq n_0$ *and all inputs of size* n, *the algorithm's running time is* $\leq c \cdot f(n)$.

- Implication 1: No need to really find the worst input.
- Implication 2: No need to consider input of size smaller than a constant n_0.

The algorithm's worst case running time is $\Omega(f(n))$

- There exists at least one input of (large) size n
  for which the running time of the algorithm is $\geq c \cdot f(n)$

Further expanded:
There exist const $c > 0, n_0 \geq 0$, *s.t. for any* $n \geq n_0$, *there exists some input of size* n *on which the algorithm's running time is* $\geq c \cdot f(n)$.

- Mainly used to show that the big-Oh analysis is tight
  (i.e., the best possible upper bound); often not required.

More on Worst-Case Analysis

Example: Insertion Sort (*n is the instance size)*

We saw that, on all inputs of size n, Insertion Sort runs in $\leq \frac{n(n-1)}{2}$ *time.*

On some inputs, e.g., if the items are in reverse order, it ***requires*** $\frac{n(n-1)}{2}$ *time, while on others, e.g., already sorted data, it only takes* n *time*

⇒ Insertion sort runs in $O(n^2)$ time (upper bound)
Insertion sort runs in $\Omega(n^2)$ time (lower bound). $\Omega(n)$ (and any function asymptotically smaller than n^2) is also a lower bound, but it is not tight

⇒ Insertion sort runs in $\Theta(n^2)$ time

Assume that you have computed $O(n^2)$ as the worst case upper bound for an algorithm A. At this point, you do not know whether the bound is tight (e.g., the worst case running time of A maybe $O(n \log n)$. Once you establish that the lower bound is $\Omega(n^2)$ then you know that $O(n^2)$ is tight and the running time of A is $\Theta(n^2)$.

Exercise 2 (from previous exam)

We have two algorithms, A and B. Let $T_A(n)$ and $T_B(n)$ denote the time complexities of algorithm A and B respectively, with respect to the input size n. Complete the last column of the following table with A, B, or U, where:

A means that for large enough n; algorithm A is always faster;
B means that for large enough n; algorithm B is always faster;
U means that the information provided is not enough to justify

Case	$T_A(n)$	$T_B(n)$	Faster
1	$\Theta(n^{1.5})$	$\Theta(n^2/(\log n)^3)$	A
2	$O(n^2)$	$\Omega(2^{\sqrt{n}})$	A (A is polynomial, B is exponential)
3	$O(\log n)$	$\Theta(2^{\log_2 \log_2 n})$	U, $m = \log_2 n, A = O(m), B = \Theta(2^{\log_2 m}) = \Theta(m)$
4	$\Theta((\log n)^3)$	$\Theta(\sqrt[3]{n})$	A (A is logarithmic, B is polynomial)
5	$O(n^4)$	$O(n^3)$	U (both are upper bounds)
6	$\Omega(n^3)$	$O(n^{2.8})$	B
7	$\Theta(n^3)$	$\Theta(4^{\log_5 n})$	B $\Theta(4^{\log_5 n}) = \Theta(n^{\log_5 4}) = O(n)$

Simple theoretical analysis is not the end of the story

Example: Selection sort, insertion sort, and wild-guess sort all have worst-case running time $\Theta(n^2)$. How to distinguish between them?

Theoretical analysis loses information because

- Asymptotic notation keeps only the largest terms and suppresses multiplicative constants
- Worst-case analysis ignores non worst-case inputs, which could be the majority.

When algorithms have the same theoretical running time differentiate via a

- Closer examination of hidden constants
- Careful analysis of typical expected inputs
- Other factors such as cache efficiency, parallelization are important
- Empirical comparison

But, theoretical analysis provides first guidelines

- Useful when you don't know what inputs to expect
- An $\Theta(n \log n)$ algorithm is (almost) always better than an $\Theta(n^2)$ algorithm, for large enough inputs
  - Will see several $\Theta(n \log n)$ sorting algorithms later

Writing down algorithms in pseudocode

- How should we describe algorithms
  - Implementable code is too hard to read
    - Also, "most" lines of code, while important in practice, are not important algorithmically and would just hide the main ideas
  - Natural language is usually not descriptive enough to concisely describe algorithmic ideas

- Will often use pseudocode
  - Tries to get across main ideas clearly. Models programming syntax but also uses natural language.
  - Not formally defined
  - Ad Hoc "rules" on next page

Insertion-Sort(A):

for $j \leftarrow 2$ **to** n **do**
    $i \leftarrow j - 1$
    while $i \geq 1$ **and** $A[i] > A[i+1]$ **do**
        swap $A[i]$ **and** $A[i+1]$
        $i \leftarrow i - 1$

Wild-Guess-Sort(A):

$\pi \leftarrow [4, 7, 1, 3, 8, 11, 5, \dots]$
check if $A[\pi[i]] \leq A[\pi[i+1]]$ **for all** $i = 1, 2, \dots, n-1$
if yes, output A **according to** π **and terminate**
else Insertion-Sort(A)

Writing down algorithms in pseudocode

- Use standard keywords (`if/then/else, while, for, repeat/until, return`) and notation: variable ← value, Array[index], function(arguments), etc.
- Indent everything carefully and consistently; may also use { } for clarity.

- Use **standard** mathematical notation and NOT programming language. E.g.

> Write i = i+1 instead of i++
> Write $x \cdot y$ and $x \bmod y$ instead of $x * y$ and $x \,\%\, y$
> Write $\sqrt{x}$ and a^b instead of $sqrt(x)$ and $power(a, b)$

- Use data structures as black boxes. If data structure is new, define its functionality first; then describe how to implement each operation.
- Use standard/learned algorithms (e.g. sorting) as black boxes
- Use functions to decompose complex algorithms.

- Use natural/plain language when it's clearer or simpler (e.g., if A is an array, you may write "$x \leftarrow$ the maximum element in A").

Exercise 3 Stirling's formula

Prove that $\log(n!) = \Theta(n \log n)$

First, I will prove that $\log(n!) = O(n \log n)$

$\log(n!) = \log(n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1) = \log(n) + \log(n-1) + \ldots + \log(1) \leq$
$\log(n) + \log(n) + \ldots + \log(n) = n \log(n) = O(n \log n)$

Then, I will prove that $\log(n!) = \Omega(n \log n)$

$\log(n!) = \log(n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1) = \log(n) + \log(n-1) + \ldots + \log(1) \geq$
$\log(n) + \log(n-1) + \ldots + \log(n/2) \geq \log(n/2) + \log(n/2) + \ldots + \log(n/2) \geq =$
$n/2 \log(n/2) = n/2 \, (\log n - \log 2) = n/2 \log(n) - \frac{n}{2} = \Omega(n \log n)$

Thus,
$\log(n!) = \Theta(n \log n)$