Algorithms 01

CS201

Kaustuv Nag

What is an Algorithm?

- An *algorithm* is any well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output*.
  - An algorithm is thus a sequence of computational steps that transform the input into the output.
  - We can also view an algorithm as a tool for solving a well-specified computational problem.
- An algorithm is said to be *correct* if, for every input instance, it halts with the correct output.
  - A correct algorithm solves the given computational problem.

Formal Definition of a Problem: An Example

Formal definition of a sorting problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

- An *instance of a problem* consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Algorithm: Characteristics

- **Finiteness:** Sequence of steps must be finite.
- **Definiteness:** Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
- **Correctness:** Must supply desired output.
- **Generality:** Should work on all possible inputs.
- **Finite resource usage:** Must terminate after finite amount of time and use only finite amount of memory.

Algorithms!

- What kinds of problems are solved by algorithms?
- "Bad programmers worry about the code. Good programmers worry about data structures and their relationships." – Linus Torvalds
- "Algorithms + Data Structures = Programs" – a 1976 book written by Niklaus Wirth covering some of the fundamental topics of computer programming.
  - A *data structure* is a way to store and organize data in order to facilitate access and modifications.
- An algorithm can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a precise description of the computational procedure to be followed.
  - We typically describe algorithms as programs written in a *pseudocode* that is similar in many respects to C, C++, Java, and Python.

An attempt to solve a problem!

Problem Statement: Consider the growth of an idealized (biologically unrealistic) rabbit population, assuming that: a newly born breeding pair of rabbits are put in a field; each breeding pair mates at the age of one month, and at the end of their second month they always produce another pair of rabbits; and rabbits never die, but continue breeding forever. How many pairs will there be in one year?

Fibonacci - version 1

```
// fib -- compute Fibonacci(n)
function fib(integer n): integer
  assert (n >= 0)
  if n == 0: return 0 fi
  if n == 1: return 1 fi

  return fib(n - 1) + fib(n - 2)
end
```

Fibonacci - version 2

```
// fib -- compute Fibonacci(n)
function fib(integer n): integer
  if n == 0 or n == 1:
    return n
  else-if f[n] != -1:
    return f[n]
  else
    f[n] = fib(n - 1) + fib(n - 2)
    return f[n]
  fi
end
```

Fibonacci - version 3

```
// fib -- compute Fibonacci(n)
function fib(integer n): integer
  if n == 0 or n == 1:
    return n
  fi
  let u := 0
  let v := 1
  for i := 2 to n:
    let t := u + v
    u := v
    v := t
  repeat
  return v
end
```

A Scheduling Problem

A set of n jobs has to be processed with 2 identical machines available onwards from time zero that can handle one job at a time. The processing of job j ($j = 1, 2, \cdots, n$) requires an uninterrupted period of p_j units of time of and it has to be executed by a single machine. Look for a feasible schedule, that is, an allocation of each job j to a time interval of length p_j on a machine such that no two jobs are processed by the same machine at the same time. The objective is to find a schedule in which the latest job finishes as soon as possible.

Insertion Sort

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

We formally denote the properties of $A[1\ldots j-1]$ as a **loop invariant:**
At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1\ldots j-1]$ consists of the elements originally in $A[1\ldots j-1]$, but in sorted order.

Loop invariants and the Correctness of Insertion Sort

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Initialization: Insertion Sort

The loop invariant holds before the first loop iteration, when $j = 2$.

- The subarray $A[1 \ldots j-1]$ therefore, consists of just the single element $A[1]$, which is the original element in $A[1]$.
- Moreover, this subarray is sorted (trivially, of course).

These arguments show that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Insertion Sort

Each iteration maintains the loop invariant.

- The body of the for loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4–7), at which point it inserts the value of $A[j]$ (line 8).
- The subarray $A[1\ldots j]$ then consists of the elements originally in $A[1\ldots j]$, but in sorted order.
- Incrementing j for the next iteration of the for loop then preserves the loop invariant.

These arguments show that the loop invariant holds prior to the first iteration of the loop.

Termination: Insertion Sort

Examine what happens when the loop terminates.

- The condition causing the for loop to terminate is that $j >= A.length = n$.
- Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time.
- Substituting $n + 1$ for j in the wording of loop invariant, we have that the subarray $A[1 \dots n]$ consists of the elements originally in $A[1 \dots n]$, but in sorted order.
- Observing that the subarray $A[1 \dots n]$ is the entire array, we conclude that the entire array is sorted.

Hence, the algorithm is correct.

Pseudocode Conventions

- **if-else** statements
- **for** loop
- **while** loop
- **repeat-until** loop
- Typically two ways of creating blocks: (i) **begin-end** and (ii) using indentation
- We assume that the loop counter retains its value after exiting the loop.
- Typically compound data is organized into *objects*, which are composed of *attributes*.
- Typically parameters are passed by values: the called procedure receives its own copy of the parameters. When objects are passed, the pointer to the data representing the object is copied, but the object's attributes are not.
- Boolean operators **"and"** and **"or"** are short circuiting.
- The keyword error indicates that an error occurred because conditions were wrong for the procedure to have been called.

Analyzing Algorithms

- Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.
  - Generally, by analyzing several candidate algorithms for a problem, we can identify the most efficient one.
- The best notion for input size depends on the problem being studied.
  - It may be the number of items in the input.
  - It may be total number of bits needed to represent the input in ordinary binary notation.
  - Sometimes, it is more appropriate to describe the size of the input using more than one numbers.
- The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed.
  - It is convenient to define the notion of step so that it is as machine-independent as possible.

Analyzing Insertion Sort

- For each $j = 2, 3, \ldots, n$, where $n = A.length$, let t_j denote the number of times the while loop test in line 5 is executed for that value of j.

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $\quad key = A[j]$	c_2	$n-1$
3 $\quad$ // Insert $A[j]$ into the sorted sequence $A[1 \mathinner{..} j-1]$.	0	$n-1$
4 $\quad i = j - 1$	c_4	$n-1$
5 $\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $\qquad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $\qquad i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $\quad A[i+1] = key$	c_8	$n-1$

Analyzing Insertion Sort

- To compute $T(n)$, the running time of insertion sort on an input of n values, we sum the products of the cost and times columns, obtaining

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- Even for inputs of a given size, an algorithm's running time may depend on which input of that size is given.
  - In insertion sort, the best case occurs if the array is already sorted and $t_j = 1$:

$$\begin{aligned} T(n) =& c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ & (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

  - We can express this running time as $an + b$ for constants a and b that depend on the statement costs c_i; it is thus a linear function of n.

Analyzing Insertion Sort

- If the array is in reverse sorted order, i.e., in decreasing order, the worst case results for insertion sort. In that case, $t_j = j$.
  - Note:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1; \quad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

  - The worst case time complexity is:

$$\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8)\,.
\end{aligned}$$

The Importance of Worst Case Analysis

- The worst-case running time of an algorithm gives us an upper bound on the running time for any input.
  - Knowing it provides a guarantee that the algorithm will never take any longer.
- For some algorithms, the worst case occurs fairly often.
  - For example, in searching a database for a particular piece of information, the searching algorithm's worst case will often occur when the information is not present in the database.
- The "average case" is often roughly as bad as the worst case.
  - Suppose, we randomly choose n numbers and apply insertion sort. On average, half the elements in $A[1 \ldots j-1]$ are less than $A[j]$, and half the elements are greater. On average, therefore, $t_j = j/2$ resulting average-case running time to be a quadratic function of the input size, just like the worst-case running time.

Design of Algorithms

- We used *incremental* approach to design insertion sort.
- We now use *divide-and-conquer* approach to design merge sort.
  - *Divide* the problem into a number of subproblems that are smaller instances of the same problem.
  - *Conquer* the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
  - *Combine* the solutions to the subproblems into the solution for the original problem.
- Merge sort using *divide-and-conquer* approach:
  - *Divide:* Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
  - *Conquer:* Sort the two subsequences recursively using merge sort.
  - *Combine:* Merge the two sorted subsequences to produce the sorted answer.

Merge - Pseudocode

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
      L[i] = A[p + i - 1]
for j = 1 to n2
      R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
      if L[i] ≤ R[j]
            A[k] = L[i]
            i = i + 1
      else A[k] = R[j]
            j = j + 1
```

Merge - Correctness

Loop Invariant

- At the start of each iteration of the for loop of lines 12–17, the subarray $A[p \dots k-1]$ contains the $k-p$ smallest elements of $L[1 \dots n_1+1]$ and $R[1 \dots n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Initialization

- Prior to the first iteration of the loop, we have $k = p$.
  - The subarray $A[p \dots k-1]$ is empty.
  - This empty subarray contains the $k-p=0$ smallest elements of L and R.
  - Since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Merge - Correctness

Loop Invariant

- At the start of each iteration of the for loop of lines 12–17, the subarray $A[p \dots k-1]$ contains the $k-p$ smallest elements of $L[1 \dots n_1+1]$ and $R[1 \dots n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Maintenance

- Suppose, $L[i] < R[j]$.
  - Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p \dots k-1]$ contains the $k-p$ smallest elements, after line 14 it copies $L[i]$ into $A[k]$, the subarray $A[p \dots k]$ will contain the $k-p+1$ smallest elements. Incrementing k (in the for loop update) and i (in line 15) reestablishes the loop invariant for the next iteration.
- If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

Merge - Correctness

Loop Invariant

- At the start of each iteration of the for loop of lines 12–17, the subarray $A[p \dots k-1]$ contains the $k-p$ smallest elements of $L[1 \dots n_1+1]$ and $R[1 \dots n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Termination

- At termination, $k = r+1$.
  - By the loop invariant, the subarray $A[p \dots k-1]$, which is $A[p \dots r]$, contains the $k-p = r-p+1$ smallest elements of $L[1 \dots n_1+1]$, in sorted order. The arrays L and R together contain $n_1+n_2+2 = r-p+3$ elements.
  - All but the two largest have been copied back into A, and these two largest elements are the sentinels.
- If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

Merge Sort - Pseudocode

- To sort the entire sequence $A = \langle A[1], A[2], \ldots, A[n] \rangle$, we make the initial call MERGE-SORT$(A, 1, A.length)$, where once again $A.length = n$.

MERGE-SORT(A, p, r)

if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT$(A, q + 1, r)$
    MERGE(A, p, q, r)

Divide and Conquer - Running Time

- When an algorithm contains a recursive call to itself, we can often describe its running time by a *recurrence equation* or *recurrence*.
- Let $T(n)$ be the running time on a problem of size n.
- If the problem size is small enough, say $n \leq c$ for some constant c, the straightforward solution takes constant time, $\Theta(1)$.
- Suppose, the division yields a subproblems, each of which is $1/b$.
- If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Merge Sort - Running Time

- For merge sort, $c = 1$, $a = 2$, $b = 2$, $D(n) = \Theta(1)$, and $C(n) = \Theta(n)$. Therefore,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

White Board

White Board