CS624 - Analysis of Algorithms

Introduction

January {23,25} 2024

Analysis of Algorithms

Goals

- Is an algorithm correct? Why?
- How “good” or “efficient” is an algorithm?
  That is, how much time and/or space does it take?

  This question does not always have clear-cut answers.
  There are subtle, important variations to this question.
- How do we adapt and compose algorithms to solve new problems?

Example: A Dictionary

Definition (Dictionary)

A **dictionary** is a set of $\langle key, value \rangle$ pairs.
Keys should be unique. Values not necessarily so.
Operation: Look up a key, retrieve the value associated with it.

Implementations:

- Some dictionaries are built once, no deletion (eg symbol tables). A hash table is the best implementation.
- Some dictionaries should handle insertions and deletions, or order is important. A binary search tree may be a better implementation.

The analysis of algorithms

- Many problems can be solved in more than one way.
- Often there is no absolutely one best algorithm.
- We need a way to reason about it, so as to have a mathematical, rigorous analysis of the performance of the algorithm and/or its correctness.
- This is often better than “it just seemed to work best” (although in some cases this is a good answer too...).
- We will do a lot of proofs in this course—both of correctness and of performance.
- First example: sorting.

Sorting

Given an array of numbers A, update A so that

Sorting

Sorting

Given an array of numbers A, update A so that

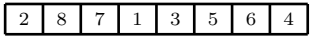
- it contains the same set of elements as before, and each element occurs the same number of times, and
- the elements occur in A in ascending order

Insertion Sort

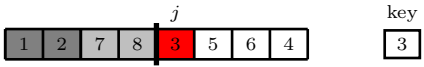
Algorithm 1 Insertion Sort

1: **for** $j \leftarrow 2$ **to** $length[A]$ **do**
2: $key \leftarrow A[j]$
3: $i \leftarrow j - 1$
4: // Insert $A[j]$ into sorted sequence $A[1..(j-1)]$
5: **while** $i > 0$ and $A[i] > key$ **do**
6: $A[i+1] \leftarrow A[i]$
7: $i \leftarrow i - 1$
8: **end while**
9: $A[i+1] \leftarrow key$
10: **end for**

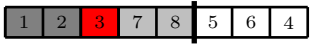
Insertion Sort: Example



The initial unsorted array.



Beginning of step j. In this case, $j = 5$. The variable key holds the value 3. All the gray elements (to the left of the thick vertical line) are in order.



End of step j. Now all the elements to the left of the thick line (which has been moved 1 element to the right) are in order. The next step will start with $j = 6$ and key will hold the value 5.

Analyzing Insertion Sort

We are interested in two things here:

- Correctness: How do we know the algorithm is correct? How can we prove it always gives the correct answer?
- Efficiency: What is its run time?

Correctness via Loop Invariants

We will use a loop invariant to prove the algorithm correct.

Definition (Loop Invariant)

A **loop invariant** is a proposition that holds at the beginning of every loop iteration and at the end of the loop.

Recipe for using a loop invariant:

- *Initialization:* You *must prove* that the loop invariant holds at the start of the loop.
- *Maintenance:* You *must prove* that if the loop invariant held at the start of iteration j, it must hold at the start of iteration $j + 1$.
- *Termination:* Immediately after the loop exits, you *can rely* on the loop invariant for the index value that causes the exit.

Correctness of Insertion Sort

Lemma (Loop Invariant for Insertion Sort)

At the start of each iteration of the loop (each iteration being characterized by a value of j), the numbers in A[1..(j-1)] are

- *the same numbers that were originally in A[1..(j-1)], and*
- *now in sorted order.*

The elements in A[j..length(A)] are unchanged.

Proof of Correctness

Proof.

Initialization:

- When j=2 this is trivially true (why?).

Maintenance:

- Assume this is true for some j. That is, A[1..(j-1)] is sorted.
- Now we must show that the execution of iteration j makes the loop invariant hold for j+1.
- The code for loop iteration j inserts the j^{th} element (key) below the last element in A[1..(j-1)] that is greater than it, so it is smaller than all the elements to its right and larger than all the elements to its left (why?), and now A[1..j] are sorted.

Termination:

- When the loop exits, the invariant holds for length[A]+1, so the entire array is sorted.

□

Comments About Proof by Induction

A loop invariant proof is a kind of **proof by induction**, and the loop invariant is the **inductive hypothesis**.

People often get very confused about what the inductive hypothesis is and how inductive proofs "work".

One may think of the inductive hypothesis as a sequence of statements. In this case, the statements are as follows:

- Statement 5 is: "At the start of iteration 5 of the loop, the numbers in A[1..4] are in sorted order."
- Statement 6 is: "At the start of iteration 6 of the loop, the numbers in A[1..5] are in sorted order."
- ...

Run Time

Question

How much time does Insertion Sort take?

On what hardware? What else is running? Is the cache hot?

To make analysis tractable, we *abstract* away hardware details. We count operations (additions, multiplications, comparisons, assignments) as units of work.

Run Time Analysis

Question

How much time does Insertion Sort take on an abstract machine?

It still depends on the input.

- The running time is expressed in the length of the array, n.
  (It is important to specify what n is.)
- Variations:
  - Worst case run time: guaranteed not to do worse than that
  - Best case run time: less practical
  - Average case run time: average over all possible inputs
    We need to know something about the distribution of possible inputs. This is the most difficult to do but often the most practical.

Best Case Run Time for Insertion Sort

The best-case time occurs when the input array is already sorted.

- In this case the while loop is never executed.
- The only cost is to test the loop condition.
- Therefore, the cost of each loop iteration is some constant c.

The runtime is therefore

$$T(n) = \sum_{j=2}^{n} c = (n-1)c$$

Worst Case Run Time for Insertion Sort

- When the array is sorted in reverse and the inner **while** loop runs $j-1$ times on the j^{th} iteration of the outer loop.
- If a is the overhead for the instructions outside the **while** loop (constant) and c is the runtime inside the **while** loop we have:

$$T(n) = \sum_{j=2}^{n} (a + (j-1)c)$$

which is of the form

$$An^2 + Bn + C$$

for some constants A, B, C.

Counting Operations

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

where $t_j(A)$ is the number of times the **while** loop test is executed
so $(t_j(A)-1)$ is the number of times the **while** loop body is executed

Counting Operations

Counting the operations as mentioned above gives the run time for the array A with length n:

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

Simplify by combining the constants:

$$T(n,A) = C_1 n + C_2 + C_3 \cdot \sum_{j=2}^{n}\left(t_j(A) - 1\right)$$

Counting Operations: The Worst Case

$T(n,A)$ is as big as possible when $t_j(A) = j$. This happens when A is initially sorted in reverse order. Since

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

we see that $T(n,A)$ is of the form $an^2 + bn + c$ — quadratic in n.

Counting Operations: The Best Case

$T(n,A)$ is as small as possible when $t_j(A) = 1$. This happens when A is initially sorted in the proper order. In this case we have

$$\sum_{j=2}^{n}(1-1) = 0$$

and so $T(n,A)$ is of the form $an + b$ — linear in n.

Average Case Run Time

What does “average case” mean?

Average over uniform distribution of all possible inputs of length n.

Except... we don’t want to count $(1, 3, 2)$ separately from $(1, 100, 50)$. They represent the same permutation of a sorted array.

Goal: Find the average, over all permutations p of size n, of $T(n, p)$.

Sub-goal: Find the average, over all permutations p of size n, of

$$\sum_{j=2}^{n} (t_j(p) - 1)$$

Difficult but not impossible!

Permutations and Inversions

We want to average over all the possible permutations of the input. Each permutation has a set of inversions.

Definition (Inversion)

An **inversion** in a sequence a of numbers $(a_1, \ldots, a_n)$ is an ordered pair (a_i, a_j) such that $i < j$ and $a_i > a_j$.

Example

How many inversions are there in the sequence $(4, 2, 3, 5, 1)$?

Permutations and Inversions

We want to average over all the possible permutations of the input. Each permutation has a set of inversions.

Definition (Inversion)

An **inversion** in a sequence a of numbers $(a_1, \ldots, a_n)$ is an ordered pair (a_i, a_j) such that $i < j$ and $a_i > a_j$.

Example

How many inversions are there in the sequence $(4, 2, 3, 5, 1)$?

Answer: 6 inversions

$\{(4,2), (4,3), (4,1), (2,1), (3,1), (5,1)\}$

$\{(a_1, a_2), (a_1, a_3), (a_1, a_5), (a_2, a_5), (a_3, a_5), (a_4, a_5)\}$

Permutations and Inversions in Insertion Sort

Lemma

*The number of inversions in the input data is exactly the number of times the inner **while** loop is executed.*

Recall the algorithm:

```
1: for j ← 2 to length[A] do
2:     key ← A[j]
3:     i ← j − 1
4:     // Insert A[j] into sorted sequence A[1..(j − 1)]
5:     while i > 0 and A[i] > key do
6:         A[i + 1] ← A[i]
7:         i ← i − 1
8:     end while
9:     A[i + 1] ← key
10: end for
```

Permutations and Inversions in Insertion Sort

Lemma

*The number of inversions in the input data is exactly the number of times the inner **while** loop is executed.*

Proof.

Consider a particular iteration of the **while** loop.

- It is uniquely identified by the values of j and i, and $i < j$.
- key is the original value of $A[j]$
- $A[i]$ was originally at $A[i']$ for some $i' < j$
- Assume $i > 0$; otherwise the loop stops.
- There are two possibilities:
  1. $A[i] > key$
  2. $A[i] \not> key$

Permutations and Inversions in Insertion Sort

Proof (continued).

We have two options:

1. $A[i] > key$
   Then $(A[i], key)$ is an inversion in the original sequence.
   We *execute* the loop body and *eliminate* the inversion (that is, we'll never move those two elements past each other again).
2. $A[i] \not> key$
   Then it does not correspond to an inversion in the original sequence, and we halt the **while** loop.

Thus on the j^{th} iteration of the **for** loop, the **while** loop is executed exactly one per inversion involving $A[j]$ and the elements to the left of it. This is true for all js, so eventually the **while** loop is executed once per inversion. □

Permutations and Inversions in Insertion Sort

Lemma

*The number of inversions in the input data is exactly the number of times the inner **while** loop is executed.*

Let me restate the run time for a permutation p of length n:

$$T(n,p) = C_1 n + C_2 + C_3 \sum_{j=2}^{n} (t_j(p) - 1)$$

where $t_j(p)$ is the number of times the **while** loop test is evaluated for a particular j iteration. The lemma above says that

$$\sum_{j=2}^{n} (t_j(p) - 1) = \text{inversions}(p)$$

So

$$T(n,p) = C_1 n + C_2 + C_3 \cdot \text{inversions}(p)$$

Permutations and Inversions in Insertion Sort

The average case run time for an input of length n is

$$\begin{aligned} T(n) &= \frac{1}{|\text{perm}(n)|} \sum_{p \in \text{perm}(n)} T(n,p) \\ &= \frac{1}{n!} \sum_{p \in \text{perm}(n)} (C_1 n + C_2 + C_3 \cdot \text{inversions}(p)) \\ &= C_1 n + C_2 + \frac{C_3}{n!} \sum_{p \in \text{perm}(n)} \text{inversions}(p) \end{aligned}$$

- $|\text{perm}(n)| = n!$
- How can we compute that sum?!
- There is a nice trick to solve this problem: compute it twice!

Total Number of Inversions

Definition (Reverse Permutation)

Given a permutation $(a_1, a_2, .., a_n)$ its **reverse permutation** $(a_n, a_{n-1}, .., a_1)$ consists of the same numbers in reverse order. Denote the reverse permutation $(b_1, b_2, ..., b_n)$ such that $b_1 = a_n$, $b_2 = a_{n-1}$ etc.

Given the permutation $(4, 2, 3, 5, 1)$, its reverse is $(1, 5, 3, 2, 4)$.

$$a_1 = 4 = b_5 \qquad a_4 = 5 = b_2$$
$$a_2 = 2 = b_4 \qquad a_5 = 1 = b_1$$
$$a_3 = 3 = b_3$$

The pair (a_i, a_j) corresponds to (b_{n-i+1}, b_{n-j+1}) in the reverse permutation. In this example, (a_1, a_2) corresponds to (b_4, b_5).

Reverse Permutations

Fact

Let $(b_1, \ldots, b_n)$ be the reverse permutation of $(a_1, \ldots, a_n)$.
Then (a_i, a_j) is an inversion iff (b_{n-i+1}, b_{n-j+1}) is not an inversion.

- How many ordered pairs (a_i, a_j) such that $i < j$ are there?

Reverse Permutations

Fact

Let $(b_1, \ldots, b_n)$ be the reverse permutation of $(a_1, \ldots, a_n)$.
Then (a_i, a_j) is an inversion iff (b_{n-i+1}, b_{n-j+1}) is not an inversion.

- How many ordered pairs (a_i, a_j) such that $i < j$ are there?
- That's $\binom{n}{2} = \frac{n(n-1)}{2}$.
- Each such pair is either an inversion in the original permutation or its reverse permutation.
- The total inversions in both permutations is therefore $\binom{n}{2}$.

Number of Inversions

$$
\begin{aligned}
&\frac{1}{n!} \sum_{p \in \text{perm}(n)} \text{inversions}(p) \\
&= \frac{1}{2n!} \sum_{p \in \text{perm}(n)} \left(\text{inversions(p)} + \text{inversions}(\overleftarrow{p})\right) \\
&= \frac{1}{2n!} \sum_{p \in \text{perm}(n)} \binom{n}{2} \\
&= \frac{1}{2n!} \cdot n! \cdot \binom{n}{2} \\
&= \frac{1}{2} \cdot \frac{n(n-1)}{2} = \frac{n(n-1)}{4}
\end{aligned}
$$

Average Case Run Time of Insertion Sort

So the average-case run time for an input of length n is

$$T(n) = C_1 n + C_2 + C_3 \frac{n(n-1)}{4}$$

It's quadratic.

Divide and Conquer

Definition (Divide and Conquer)

A **divide and conquer** algorithm consists of three parts:

1. Divide the problem into smaller sub-problems.
2. Recursively solve each sub-problem *independently*.
3. Combine the solutions to solve the original problem.

The cost of parts 1 and 3 must be relatively cheap.

For this to work we must have a way to divide the problem so that the solutions of the sub-problems are related to the overall solution.

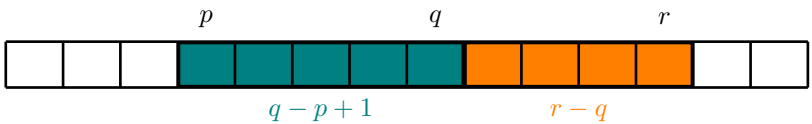
Merge Sort is an example of a divide and conquer algorithm.

Merge Sort

- Input: An array A[1..n] of numbers.
- MergeSort(A,p,r) sorts all the elements in positions p..r.
  Call MergeSort(A,1,n) to sort the entire array.

Algorithm 2 MergeSort(A,p,r)

1: **if** $p < r$ **then**
2: $q \leftarrow \lfloor (p+r)/2 \rfloor$
3: $MergeSort(A, p, q)$
4: $MergeSort(A, q+1, r)$
5: $Merge(A, p, q, r)$
6: **end if**



The Merge Subroutine

Algorithm 3 Merge(A,p,q,r)

1: $n_1 \leftarrow q - p + 1$
2: $n_2 \leftarrow r - q$
3: // Create arrays $L[1...n_1 + 1]$ and $R[1...n_2 + 1]$
4: **for** $i \leftarrow 1$ to n_1 **do**
5: $\quad L[i] \leftarrow A[p + i - 1]$
6: **end for**
7: **for** $j \leftarrow 1$ to n_2 **do**
8: $\quad R[j] \leftarrow A[q + j]$
9: **end for**
10: $L[n_1 + 1] \leftarrow \infty$
11: $R[n_2 + 1] \leftarrow \infty$
12: $i \leftarrow 1; j \leftarrow 1$
13: **for** $k \leftarrow p$ **to** r **do**
14: $\quad$ **if** $L[i] \leq R[j]$ **then**
15: $\quad\quad A[k] \leftarrow L[i]$
16: $\quad\quad i \leftarrow i + 1$
17: $\quad$ **else**
18: $\quad\quad A[k] \leftarrow R[j]$
19: $\quad\quad j \leftarrow j + 1$
20: $\quad$ **end if**
21: **end for**

Correctness of Merge

Definitions (Precondition and Postcondition)

- A **precondition** is a proposition that must be true on entry to a function or at the beginning of a section of code.
- A **postcondition** is a proposition that is true on exit from a function or at the end of a section of code.

Merge

Concerning a call to $Merge(A, p, q, r)$:

- Precondition: $A[p \,..\, q]$ and $A[(q+1) \,..\, r]$ must both be sorted.
- Postcondition: $A[p \,..\, r]$ is sorted.

If Merge is called without the precondition holding, there is no hope that it will behave correctly.

Correctness of Merge

We'll use a loop invariant to prove the correctness of Merge.

Lemma (Loop Invariant for Merge)

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

Correctness of Merge

Proof.

Initialization

- When $k = p$, this is vacuously true. (Why?)

Maintenance

- Assume the loop invariant holds for some k. Now we must show that the execution of iteration k makes the loop invariant hold for $k + 1$.
- There are two possibilities:
  1. $L[i] \leq R[j]$.

     Then $L[i]$ is the smallest element not yet copied back into A. So we set $A[k] \leftarrow L[i]$, and now $A[p \,..\, k]$ contain the $(k + 1) - p$ smallest elements.

     Now $A[p \,..\, k]$ contains $L[i]$, so we increment i to restore the second part of the loop invariant.
  2. $L[i] > R[j]$. Similar.

Termination

- When the loop exits, $k = r + 1$, so $A[p \,..\, r]$ is sorted.

□

Correctness of Merge Sort

Merge Sort

After a call to $MergeSort(A, p, r)$:

- Postcondition: The array section $A[p..r]$ is sorted.

Proof.

The MergeSort algorithm has two cases:

1. $p < r$. We divide the section into subproblems, recur, and Merge. The subproblem postconditions satisfy the Merge preconditions, and the Merge postcondition implies the overall correctness.
2. Otherwise, the section has length 0 or 1, trivially sorted.

□

Run Time of Merge Sort

Let $T(n)$ be the run time of MergeSort. There are two cases:

- Base case ($n \leq 1$):
  Constant time, d.
- Otherwise, $T(n)$ is the sum of the three steps:
  1. Divide: Find the middle of the array.
     Constant time c.
  2. Conquer: Solve recursively for each half of the array.
     Sub-problems take $2 \cdot T(\frac{n}{2})$.
  3. Merge: Combine the two sub-arrays.
     Linear in n. Suppose it is $c \cdot n$.

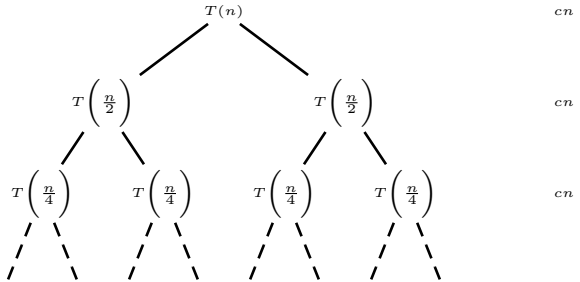
  A sufficiently large c can be used for (1) and (3).

Recurrence Formula

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

- Recurrences like this come up frequently in algorithmic analysis.
- It's important to have ways of solving them. We'll see a couple. One basic way is to form a *recursion tree*.

Recursion Tree for MergeSort



If $n = 2^p$ then there are p rows with cn on the right, and one last row with dn on the right. Since $p = \log n$, this means that the total cost is $cn \log n + dn$. In other words, this is what we call an "$O(n \log n)$" algorithm.