CS624 - Analysis of Algorithms

Introduction

September 1, 2019

Contact Information

- Instructor: Nurit Haspel
- `http://www.cs.umb.edu/~nurith`
- nurith@cs.umb.edu or nurit.haspel@umb.edu
- Phone – 617-287-6414.
- Office – S-3-071 (until the move), M-201-04 (afterwards)
- Office hours – Tu Th 2:30-4:00 or by appointment.
- Course schedule:
  - Tu Th 11:00–12:15 at S-3-143.

Course Description

- Methods for analyzing and developing algorithms.
- Runtime analysis and growth rate.
- Sorting and searching.
- Dynamic programming.
- Greedy algorithms.
- Amortized analysis.
- Graph algorithms and their applications.
- P=NP? and approximation algorithms.
- More topics if time permits.
- Course website: `http://www.cs.umb.edu/cs624`.
- Syllabus.

Course Requirements

- Prerequisite: CS220 or equivalent (formerly known as CS320).
- Consider dropping if you didn't do well in CS220.
- Homework assignments – approximately every week (20% total).
- The homework due date is strict. No late assignments will be accepted without a good reason.
- There will be no programming assignment (although programming experience will definitely help!).
- You may consult with your friends, but the final work should be individual or with a small group if applicable.
- I strongly prefer typed homework. If handwritten – make it CLEAR.

Course Requirements (Cont.)

- Two quizzes (20% each), final exam (40%).
- Your final grade should be at least C (60%) to pass.
- Books:
  - Required: Introduction to Algorithms, 3rd Edition by Cormen, Leiserson, Rivest, and Stein, MIT press 2009. (you can order on amazon or get an older edition, but be sure to sync the page numbers).
  - Not required but highly recommended: The Algorithm Design Manual, 2nd Edition by Steven S. Skiena, Springer Verlag, 2008.
- You will be asked to read from the books in preparation for class.

General Stuff

- The course material will be available online and updated regularly with class notes and assignments.
- Attendance is not required (but highly encouraged). You are responsible for updating yourselves if you miss a class.
- Don't be afraid to ask questions in or out of class. I won't think you are stupid and it won't lower your grade.
- Don't hesitate to send me e-mails. I expect e-mails. It won't lower your grade.

General Stuff

- Read here about my plagiarism and cheating policy.
- The homework assignments and tests
- I have a second-strike policy.
- First strike – you get a 0 in the homework and a warning.
- Second strike – you fail the course + a report to the higher administration
- The no. of strikes applies other courses as well.
- **It is easier to catch plagiarism than you think**
- Your grade is not negotiable! The only way I will change your grade is if you prove to me I made a mistake.
- I don't give bonuses and options for extra credits on an individual basis. Your grade is based on your coursework only.

Analysis of Algorithms

- This course focuses on methods to analyze how "good" or "efficient" an algorithm is.
- This is not always a clear-cut question with a clear-cut answer.
- How much time and/or space does an algorithm take?
- There are some subtle but important variations to this question.

Example – a Dictionary

- Dictionary – a set of $< key, value >$ pairs.
- Look up a key, retrieve the value associated with it.
- Keys should be unique. Values not necessarily so.
- Some dictionaries are built once, no deletion (e.g. – 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 data structure.

The analysis of algorithms

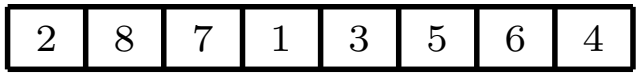
- Many problems can be solved in more than one way.
- Often there is no absolutely one best algorithm.
- We need a way to reason about it, so as to have a mathematical, rigorous analysis of the performance of the algorithm and/or its correctness.
- This is often better than "it just seemed to work best" (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.

Insertion Sort

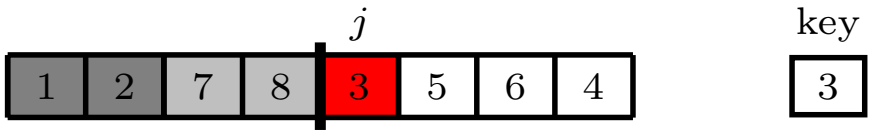
Algorithm 1 Insertion Sort

```
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
```

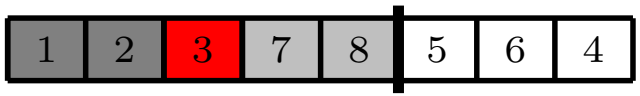
Insertion Sort - Example



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.

Insertion Sort

We are interested in two things here:

- Proof of correctness – How do we know the algorithm is correct? How can we prove it always gives the correct answer?
- Efficiency – what is its runtime?

Proof of Correctness

Proof by induction on the external for loop.

Lemma

- *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 in sorted order.*
- *In fact, they are the numbers that were originally in A[1..(j-1)], but they are now in sorted order.*

Proof of Correctness

Proof.

- When j=2 this is trivially true (why?).
- Assume this is true for some j-1.
- To pass from j-1 to j, we insert 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 is that?), and now A[1..j] are in sorted order.

Thus when we are done, the entire array is sorted. □

Comments About Proof by Induction

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."
- ...

Comments About Proof by Induction

- Obviously, statement 1 is meaningless and statement 2 is trivially true, so we have a starting point.
- The proof then showed that if for some number j, Statement j was known to be true, then it follows that the next statement – Statement j+1, must be true.
- Since we know that Statement 2 is true, it thus follows that Statement 3 must be true.
- The same reasoning shows us that we can continue in this manner and conclude that Statement j is true for all values of $j \geq 2$, which is what we needed to show.

Runtime Analysis

- The running time is expressed in the length of the array, n.
- It is important to specify what n is.
- Worst case runtime (guaranteed not to do worse than that...).
- Best case runtime (less practical).
- Average case runtime – average the runtime 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.

Runtime Analysis

- In this course we don't care about the machine specifications, which may alter the practical runtime.
- In real life we often care about those things.
- We care about the number of operations – additions, multiplications, assignments.
- We will assume they are all equally expensive (not necessarily always true, but good enough for our purposes).

Best Case Runtime Analysis for Insertion Sort

- The best-case time occurs when the input array is already sorted in the correct order.
- In this case the while loop is never executed.
- The only cost is to test the loop condition.
- Therefore, the cost of each j loop is some constant c.
- The runtime is therefore

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

Worst Case Runtime Analysis 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.

A Count of 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	$\sum_{j=2}^{n} t_j$
$A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Counting the Operations – Worst Case

Counting the operations as mentioned above gives:

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

$T(n)$ is as big as possible when $t_j = j$. 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 $T(n)$ is of the form $an^2 + bn + c$ – quadratic in n.

Counting the Operations – Best Case

$T(n)$ is as small as possible when $t_j = 1$. 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 $T(n)$ is of the form $an + b$ – linear in n.

Average Case Runtime

- It seems much more difficult since we have to average over all possible inputs.
- In other words – find the average value of $\sum_{j=2}^{n}(t_j - 1)$ over all possible permutations of the input.
- Difficult but not impossible!

Permutations and Inversions

- We want to average over all the possible permutations of the input.
- With each permutation there is a set of inversions.
- Definition: An inversion of a sequence a of numbers $(a_1..a_n)$ is an ordered pair (a_i, a_j) such that $i < j$ and $a_i > a_j$
- In other words – an inversion is a pair whose indices are in order but values out of order.
- For example – how many inversions are there in the sequence $(5, 2, 3, 7, 1)$?

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.

- At the top of the while loop we have to decide whether A[i] is larger than the key (A[j]) or not.
- The inner loop starts from j-1 and goes down.
- j is the original index.
- i may change but its current and original index are smaller than j.
- Therefore $i < j$.

Proof (Cont.)

We have two options:

1. $A[i] < A[j]$. In this case (A[i],A[j]) is not an inversion in the current or the original sequence – we break the while loop.
2. $A[i] > A[j]$. In this case (A[i],A[j]) is an inversion in the current and the original sequence. In this case we execute the while loop which gets rid of the inversion for us.

- 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 j's, so eventually the while loop is executed once per inversion.

Permutations and Inversions in Insertion Sort

- In other words – the total number of inversions in the original sequence is $\sum_{j=2}^{n}(t_j - 1)$.
- To average over all the permutations we have to sum the total number of inversions in all the permutations and divide by n! (the total number of permutations).
- There is a nice trick to solve this problem.

Total Number of Inversions

- 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.

For example given the permutation $(5, 2, 3, 7, 1)$, its reverse is $(1, 7, 3, 2, 5)$. Here we have

$$a_1 = 5 = b_5$$
$$a_2 = 2 = b_4$$
$$a_3 = 3 = b_3$$
$$a_4 = 7 = b_2$$
$$a_5 = 1 = b_1$$

Reverse Permutations

- The pair (a_i, a_j) corresponds to (b_{n-i+1}, b_{n-j+1}) in the reverse permutation.
- If we stick to the right order, the pair (a_i, a_j) corresponds to (b_{n-i+1}, b_{n-j+1}).
- In the example above – (a_1, a_2) corresponds to (b_4, b_5).
- Notice that (a_i, a_j) is an inversion iff (b_{n-i+1}, b_{n-j+1}) is not an inversion.

Number of Inversions

- How many ordered pairs (a_i, a_j) such that $i < j$ are there in the original sequence?
- Exactly the number of ways to choose 2 elements out of n – $\binom{n}{2}$.
- Each one of those is either an inversion in the original sequence or in its reverse.
- The number of inversions in both sequences is therefore $\binom{n}{2}$.

Number of Inversions

In the example we used there are $\binom{5}{2} = 10$ inversions.

(a_1, a_2)	(a_1, a_3)	(a_1, a_4)	(a_1, a_5)	(5,2)	(5,3)	(5,7)	(5,1)
	(a_2, a_3)	(a_2, a_4)	(a_2, a_5)		(2,3)	(2,7)	(2,1)
		(a_3, a_4)	(a_3, a_5)			(3,7)	(3,1)
			(a_4, a_5)				(7,1)

Number of Inversions

- If we pair every permutation and its reverse, we get exactly $\binom{n}{2}$ inversions for a pair.
- The total number of inversion is therefore $\frac{n!}{2} * \binom{n}{2}$.
- The average number of inversions in a permutation is the above divided by $n!$, the number of permutations.
- The average value of T(n) is then $\frac{1}{2} * \binom{n}{2} = \frac{n(n-1)}{4}$, which is quadratic in n.

Example – Merge Sort

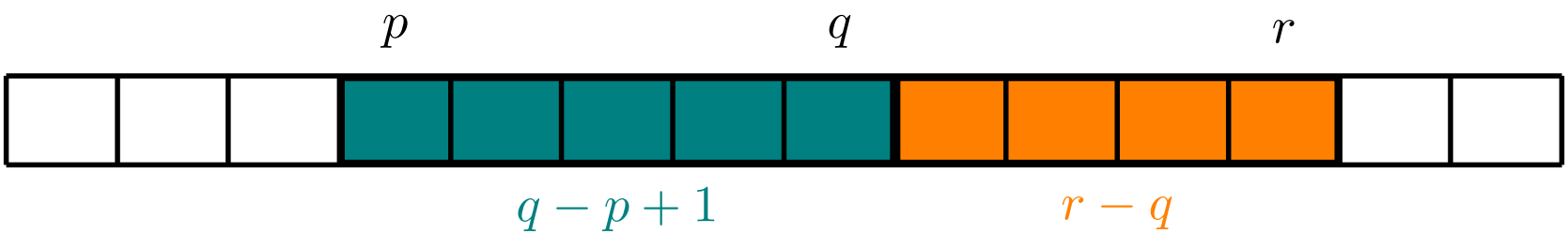
- An example of a divide and conquer algorithm:
  1. Divide the problem into smaller sub-problems.
  2. Solve each one recursively.
  3. Combine the solutions to solve the original problem.
- For this to work we must:
  - Have a way to divide the problem in a way that the solutions of the two sub-parts are related to the overall solution.
  - The cost of 1 and 3 must be relatively cheap.

Merge Sort

- Input: An array A[1..n] of numbers.
- The top level call: MergeSort(A,1,n)
- In general MergeSort(A,p,r) sorts all the elements in positions p..r.

Algorithm 2 MergeSort(A,p,r)

1: **if** $p < r$ **then**
2: $q = (p + r)/2$
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)

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
// Create arrays $L[1...n_1 + 1]$ and $R[1...n_2 + 1]$
for $i \leftarrow 1$ to n_1 **do**
    $L[i] \leftarrow A[p + i - 1]$
end for
for $j \leftarrow 1$ to n_2 **do**
    $R[j] \leftarrow A[q + j]$
end for
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1; j \leftarrow 1$
for $k \leftarrow p$ to r **do**
    if $L[i] \leq R[j]$ **then**
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    else
        $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
    end if
end for

Proof of Correctness

Here's our loop invariant:

Lemma

- *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.*

Proof of Correctness

Proof.

- When $k = p$, this is vacuously true. (Why?)
- To go from $k - 1$ to k, there are two possibilities: either $L[i] \leq R[j]$ or $L[i] > R[j]$.
- In the first case, $L[i]$ is the smallest element not yet copied back into A. (Why is this?) So we copy it into A (at the right position), and the loop invariant is maintained.
- The second case is similar.

And that finishes the proof of correctness, because when we're done, k is $r + 1$. □

Divide and Conquer

The overall runtime of MergeSort $T(n)$ is the sum of the following:

1. Divide: Find the middle of the array. Done in constant time c.
2. Conquer: Solve recursively for each half of the array. $2 * T(\frac{n}{2})$.
3. Merge: Combine the two sub-arrays. Linear in n. Suppose it is $c * n$.

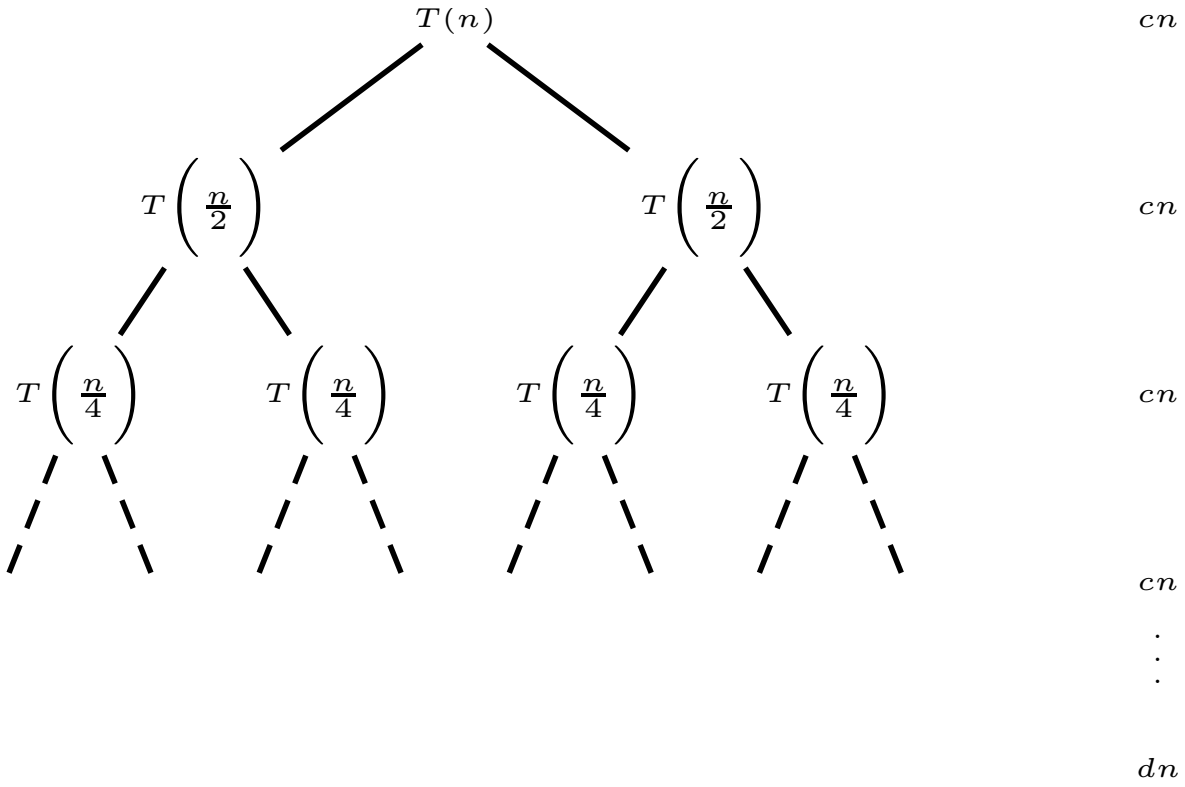
A sufficiently large c can be used for (1) and (3). Base case – one element in the array. Constant time d.

Recurrence Formula

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

- Identities 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 $cNlogN + dN$.
In other words, this is what we call an "$O(n \log n)$" algorithm.