CS 430 - Introduction to Algorithms Notes

Len Washington III

September 12, 2023

Contents

List of Algorithms

Opening Questions

1. Algorithm A takes 5 seconds to sort 1000 records, and Algorithm B takes 10 seconds to sort 1000 records. You have the code for both algorithms. When deciding which algorithm to use to sort up to 1,000,000 records, why might Algorithm B be the better choice? Algorithm B might grow slower than algorithm A when more data is added.

2. Why is it helpful to sometimes define a problem in its most basic mathematical terms? Understanding the problem space might help with thinking of algorithmic solutions since there might be a possible shortcut rather than the brute force algorithm. It also helps to know if the problem is easy, which might mean that the brute force method would work effectively.

3. In your own words, explain what a loop invariant is. A statement that's true for every iteration of a loop. It never changes. It's a way to prove an iterative algorithm correctly.

4. What are the three kinds of growth in run time analysis we may do on an algorithm. Best case, worst case, average case.

5. For recursive algorithms, what do we need to define and solve to do the runtime analysis? Recurrence relation. Base recursive?

Sorting as a Case Study for Algorithms Analysis

1. Write pseudocode for one of these iterative sorts: InsertionSort, BubbleSort, SelectionSort. Then draw pictures for a sample run on 5 random numbers showing comparisons/swaps.

2. Write the loop invariant for your sort and prove your sort is correct by proving Initialization, Maintenance, and Termination.
   An important property of sorting algorithms is whether or not it is stable: numbers with the same value appear in the output array in the same order as they do in the input array.
   Before the i^{th} iteration, all of the items that were originally before position i are now sorted. After the i^{th} iteration, the original items $1 \rightarrow i$ are sorted (that doesn't mean they're in their final position)

   Initialization: Before the $i = 1$ iteration, $A[0]$ is sorted.
   Maintenance: Line 2 correctly sets the current starting position, lines 3 and 4 verifies the element is in the current position and walks down the item.
   Termination: Before the $i = n$ iteration, which doesn't exist, all items before position n are now sorted.

3. Is your sort stable? (Is the relative order of duplicate values maintained when the algorithm is done?) This is stable because in line 3, the values are only swapped if one value is less than the other. But if the values are the same, the swap won't occur.
   Resource (memory or runtime) Use Analysis – Resource use analysis usually depends on the size of the input to the algorithm. You can write a function T(n) that matches the behavior of the resource use of the algorithm.NOTE: For recursive algorithms we develop and solve a recurrence relation to find the T(n), the resource use function.

4. For the iterative sort you wrote above, construct a run time analysis function T(n) by assigning a different constant (c1, c2, c3, etc) to each type of statement (the run time for statements of that type), and counting how many times each statement executes for an input size n. Then sum up the constants times the execution counts. You may need to define variables other than n if there are event controlled iterations with an unknown number of loops.

$$T(n) = nc_1 + (n-1)c_2 + c_3 \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} (t_i - 1)\left[c_4 + c_2\right]$$

   where t_i depends on initial order of the array. Worst case: opposite order where $t_i = i$. Best case: if the array is already sorted where $t_i = 1$.

Table 1: Constants

Line #	Constant	Number of Executions
1	c_1	n
2	c_2	$n-1$
3	c_3	t_i
4	c_4	
5	c_5	

t_i is how far the i^{th} item needs to walk ($0 \rightarrow (i-1)$)

For a more descriptive analysis we need to consider the various generic execution flows and input structures that result in those generic execution flows. There are 3 more descriptive

resource use analyses: 1) worst case (usually used) 2) average case (sometimes used) 3) best case (hardly ever used)

5. For your T(n) function from above determine the best case, worst case and average case T(n)s.Best Case ($t_i = 1$):

$$\begin{aligned} T(n) &= nc_1 + (n-1)c_2 + c_3 \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} (t_i - 1)\,[c_4 + c_2] \\ &= nc_1 + (n-1)c_2 + c_3 \sum_{i=1}^{n-1} (1) + \sum_{i=1}^{n-1} (1-1)\,[c_4 + c_2] \\ &= nc_1 + (n-1)c_2 + c_3(n-1) + \sum_{i=1}^{n-1} (0)\,[c_4 + c_2] \\ &= nc_1 + (n-1)c_2 + c_3(n-1) \\ &= nc_1 + (n-1)(c_2 + c_3) \end{aligned}$$

Worst Case ($t_i = i$):

$$\begin{aligned} T(n) &= nc_1 + (n-1)c_2 + c_3 \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} (t_i - 1)\,[c_4 + c_2] \\ &= nc_1 + (n-1)c_2 + c_3 \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} (i-1)\,[c_4 + c_2] \end{aligned}$$

Asymptotic Analysis

- To simplify comparing the resource usage of different algorithms for the same problem.
- Ignore machine dependent constants; look at the growth of $T(n)$ as $n \to \infty$.
- As you double n, what does $T(n)$ do?? Double?? Square??

Theta (Θ) Notation (more details in future lectures)

- Drop lower order terms;
- Ignore leading constants
- Concentrates on the growth
- Tight bound on growth

Ω is lowerbound, O (read as Big-O) is upperbound

1. For your best case, average cast, worst case $T(n)$ functions from above, give the asymptotic function (Θ notation)

$$T(n) = c_1 n + c_2(n-1) + c_3 \sum_{i=1}^{n-1} t_i + c_4 \sum_{1}^{n-1} (t_i - 1)$$

Worst case: $t_i = i \rightarrow O(n^2)$

Best case: $t_i = i \rightarrow O(n)$

2. Given the problem sizes and worst case runtime for one of the problem sizes, and what you know about each algorithm, predict the missing runtimes.

	$n = 100$	$n = 200$	$n = 400$	$n = 800$
Linear search $O(n)$	10 seconds	20	40	80
Binary search $O(\lg n)$	7	8 seconds	9	10
Insertion Sort $O(n^2)$	20	80	320 seconds	1,280

For Binary search, the steps increase logarithmically, meaning that for each iteration, the solution is cut in half. This means that (starting with $n = 200$), after one iteration the size has been cut in half, which would give you the $n = 100$ problem. So the time would only increase with 1 iteration for doubling the size. The 1-second increase was an estimation for how long each iteration was.

Opening Questions - Average Case Runtime

3. How did we approach case runtime analysis of iterative algorithms previously? How can we improve on this? Insertion sort – more or less, the inner loop needs to walk half way down on every other iteration.

Expectation of a Random Variable

A random variable is a variable that maps an outcome of a random process to a number. Examples:

- Flipping a coin. If heads: $X = 1$, if tails: $X = 0$
- Y =sum of 7 rolls of a fair die (there is only 1-way to get a sum of 7, get 1's on every roll. But there are multiple ways you could get a sum of 12, so this would be a random variable.)
- Z =in insertion sort, the number of swaps needed to move the ith item to its correct position in items 1 through $(i - 1)$. Range$(Z) = [0, i - 1]$ (assuming all equally likely)

The expected value of a random variable X is the sum over all outcomes of the value of the outcome times the probability of the outcome.

$$E(X) = \sum_{s \in S} X(s)\, p(s)$$

- s is the outcome
- X is the random variable (assigning a number to a certain outcome.)

- p is the probability

4. What is the expected outcome when you roll a fair die once? What about a loaded die where the probability of a side coming up is the value of the side divided by 21?

$$1\left(\frac{1}{6}\right)+2\left(\frac{1}{6}\right)+3\left(\frac{1}{6}\right)+4\left(\frac{1}{6}\right)+5\left(\frac{1}{6}\right)+6\left(\frac{1}{6}\right)=\frac{21}{6}=3.5$$

$$1\left(\frac{1}{21}\right)+2\left(\frac{2}{21}\right)+3\left(\frac{3}{21}\right)+4\left(\frac{4}{21}\right)+5\left(\frac{5}{21}\right)+6\left(\frac{6}{21}\right)=\frac{1+4+9+16+25+36}{21}=\frac{91}{21}=\frac{13}{3}=4.\bar{3}$$

5. Calculate the expected outcome when you roll a fair die twice and sum the results. Do this two different ways. There are 36 possible combinations of rolls. $2=(\{1,1\})=\frac{1}{36}, 3=(\{1,2\};\{2,1\})=\frac{2}{36},\ldots$

Now let's use expectation of a random variable to improve our average case runtime for insertion sort (similar for bubble sort or selection sort).

- Sort n distinct elements using insertion sort
- X_i is the random variable equal to the number of comparisons used to insert a_i into the proper position after the first $i-1$ elements have already been sorted. $1 \leq X_i \leq i-1$

$E(X_i)$ is the expected number of comparisons to insert a_i into the proper position after the first $i-1$ elements have been sorted.
$E(X) = E(X_2) + E(X_3) + \cdots + E(X_n)$ is the expected number of comparisons to complete the sort (our new average case runtime function).

6. Write equations for the following and simplify.

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

- $E(X_i)$Outcomes: $1,2,3,\ldots,(i-1)$, Probability for each: $\frac{1}{i-1}$

$$\begin{aligned}
E(X_i) &= \left(\frac{1}{i-1}\right)1+\left(\frac{1}{i-1}\right)2+\left(\frac{1}{i-1}\right)3+\cdots+\left(\frac{1}{i-1}\right)(i-1)\\
&= \left(\frac{1}{i-1}\right)\sum_{j=1}^{i-1} j\\
&= \left(\frac{1}{i-1}\right)\left[\frac{(i)(i+1)}{2}-i\right]\\
&= \frac{(i)(i+1)}{2(i-1)}-\frac{i}{i-1}
\end{aligned}$$

- $E(X)$

$$\sum_{i=2}^{n}\left(\frac{i}{2}\right) = O(n^2)$$

7. What if the data is not random?

Asymptotic Analysis (more details)

BIG-O Notation – Upper bound on growth of a runtime function

- $f(n) \in O(g(n))$ reads as "f(n) is big-O of g(n)"

If there exists C, n_0 such that $0 \leq f(n) \leq CG(n)$ when $n \geq n_0$ and $c > 0$.

1a. Use the definition of big-O to show $2n^2$ is big-O n^3 (find a C and n_0 that works in the above)

$$f(n) < cg(n)$$
$$2n^2 \leq cn^3 \text{ when } n > n_0 (n > 1) \text{ is } 2n^2 < cn^3$$
$$2(2^2) \leq c(2^3)$$

1b. Use the definition of big-O to show $T(n) = 3n^3 - 4n^2 + 3\lg n - n = O(n^3)$

$$0 < 3n^3 - 4n^2 + 3\lg n - n < cn^3 (\text{ find } c > 0 \text{ and } n_0 > 0 \text{ such that } n > n_0)$$
$$3n^3 - 4n^2 + 3\lg n - n? < 3n^3$$
$$-4n^2 + 3\lg n - n? < 0$$
$$\forall n > 0, -4n^2 \text{ is negative}$$

Note: The ? < means is this true

Omega Ω Notation – Lower Bound $g(n) = \Omega(g(n))$
$0 \leq cg(n) \leq f(n)$
$c?$ $n_0 < n$

2. Use the definition of omega to show $n^{\frac{1}{2}} = \Omega(\log n)$

$$n > n_0, c > 0$$
$$c\lg(n) \leq n^{\frac{1}{2}}$$
$$n = 4, c\log_2(4)? \leq 4^{\frac{1}{2}}$$
$$2c \leq 2$$
$$n = 16, c\log_2(16)? \leq 16^{\frac{1}{2}}$$
$$4c \leq 4$$
$$2^{clog_2 n} \leq 2^{\sqrt{n}}$$
$$2^c 2^{log_2 n} \leq 2^{\sqrt{n}}$$
$$n2^c \leq 2^{\sqrt{n}}$$

Theta θ Notation – Strict Bound $f(n) = \theta(g(n))$
$c_1 g(n) \leq f(n) \leq c_2 g(n)$
$c_1? < c_2?$

3a. Use the definition of theta to show $3n^3 - 4n^2 + 37n = \theta(n^3)$

$$c_1 n^3 \leq 3n^3 - 4n^2 + 37n \quad \leq c_2 n^3$$
$$2n^3? \leq 3n^3 - 4n^2 + 37n$$
$$0? \leq n^3 - 4n^2 + 37n$$
$$\text{True for } n > 0$$
$$3n^3 - 4n^2 + 37n \leq c_2 n^3$$
$$3n^3 - 4n^2 + 37n \leq 3n^3$$
$$-4n^2 + 37n \leq 0$$
$$n(-4n + 37) \leq 0 \; \forall n \geq 10$$
$$c_1 = 2, n_1 = 0, c_2 = 3, n_2 = 10$$

3b. Use the definition of theta to show $n^2 + 3n^3 = \theta(n^3)$

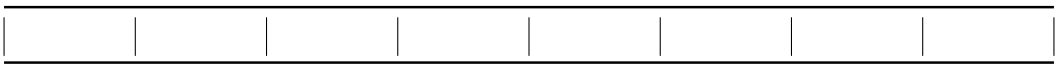
$$c_1 n^3 \leq n^2 + 3n^3 \quad \leq c_2 n^3$$
$$c_1 n \leq 1 + 3n \leq \quad c_2 n$$
$$c_1 = 2 \quad c_2 = 4 \; \forall n > 1$$

Recursive Sorting – Mergesort

- divide and conquer (and combine) approach, recursive algorithm
- key idea: you can merge sorted lists of total length n in $\theta(n)$ linear time
- base case: a list of length one element is sorted
- For all sorting algorithms, the base case is $n = 1$.

1. Demonstrate how you can merge two sorted sub-lists total n items with n compares/copies. How much memory do we need to do this? Write pseudocode to do this.

2 3 7 8 1 4 5 6



Algorithm 1 Merge algorithm

```
1:  function MERGE(A, i, j, k) ▷ i is the index of the lower sorted sequence, j is the index of the upper sorted, k is the end
2:      B ← array of size k
3:      l ← 0
4:      i_end ← i-1
5:      while i < i_end && j ≤ k do
6:          if A[i] < A[j] then
7:              B[l] ← A[i]
8:              i++
9:              l++
10:         else
11:             B[l] ← A[j]
12:             j++
13:             l++
14:         end if
15:     end while
16: end function
```

Algorithm 2 Merge sort algorithm

```
1: function MERGESORT(A, p, r)                    ▷ Initial call Mergesort(A, 1, n)
2:     if p < r then
3:         q ← (p+r)/2                            ▷ Integer division
4:         MergeSort(A, p, q)                     ▷ Recursively sort 1st half
5:         MergeSort(A, q + 1, r)                 ▷ Recursively sort 2nd half
6:         Merge(A, p, q, r)                      ▷ Merge 2 sorted sub-lists
7:     end if
8: end function
```

2. Demonstrate Mergesort on this data:

	3	41	52	26	38	57	09	49
1.	3(p=1, q=4)	41	52	26	38	57	09	49(
2.	3(p=1, q=2)	41	52	26(r=4)				
3.	3(p=1, q=1)	41 (r=2)						

I don't even know how I can write this part in LaTeX

1. A recurrence relation describes runtime function recursively for a recursive algorithm. Write a recurrence relation for the Merge sort algorithm. HINT: try to count the number of executions of each statement and the cost of each Recurrence relations for merge sort lines:

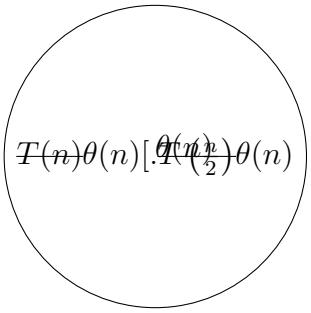
$2 = c_1, 3 = c_2, 4 = 5 = T\left(\frac{n}{2}\right)$ and $O(n)$ as the runtime for merge

$$T(n) = c_1 + c_2 + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n)$$
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Solving Recurrence Relations – Recurrence Tree Method

We solve a recurrence relation to get a function in its closed (non-recursive) form. The recurrence tree method is a visual method of repeatedly substituting in the recurrence relation for $T(n)$ on smaller and smaller n until you reach the base case, and then summing up all the nodes in the tree.

2. Draw the recurrence tree for Mergesort $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$, $T(1) = O(1)$



Divide and Conquer Algorithms

- Divide – divide the problem into sub-problems that can be solved independently
- Conquer – recursively solve each sub-problem
- Combine – possibly necessary, combine solutions into sub-problems

Not all problems can be solved with the divide and conquer approach. Maybe sub-problems are not independent, or solutions to sub-problems cannot be combined to find solution to main problem.

3. Write a recursive algorithm for Binary Search. Write and solve its recurrence relation.Divide & Conquer & Combine Runtime analysis:

Algorithm 3 Binary Search Algorithm

```
1: function BS(A,key,i, j)                    ▷ Initial call: BS(A (sorted), key, 1, n)
2:     if i ≤ j then                          ▷ Handle Base case not found
3:         k ← ⌊(i+j)/2⌋
4:         if A[k] == key thenreturn key
5:         end if
6:     end if
7: end function
```

$$T(n) = O(1) + T\left(\frac{n}{2}\right)$$
$$T(1) = O(1)$$

4. Write a recursive algorithm for Selection Sort (or insertion sort or bubble sort). Write and solve its recurrence relation.

Algorithm 4 Selection Sort Algorithm

```
1: function SELECT(A, i, j)        ▷ Not a stable sorting algorithm. Initial call: Select(A, 1, n)
2:     if i < j then                                                  ▷ Base case, 1 item is sorted.
3:         minSoFar = A[i]
4:         kk=i
5:         for k=i+1; k≤j; k++ do
6:             if A[k] < minSoFar then
7:                 minSoFar ← A[k]
8:                 kk ← k
9:             end if
10:        end for
11:        Swap A[i] with A[k]
12:        Select(A, i+1, j)
13:    end if
14: end function
```

Runtime analysis: $T(n) = T(n-1) + O(n)$, $T(1) = O(1)$

5. Describe an efficient divide and conquer algorithm to count the number of times a character appears in a string of length n.

Algorithm 5 Count of char in string

```
1: function FIND_COUNT(S, from, to, char)
2:     if from < to then
3:         mid ← (from+to)/2
4:         x ← Find_Count(A, from, mid, char)
5:         y ← Find_Count(A, mid+1, to, char)return x + y
6:     else                                                        ▷ 1 character left
7:         if S[from]==char thenreturn 1
8:         elsereturn 0
9:         end if
10:    end if
11: end function
```

Inductive Proofs

(needed in next lecture to prove a solution to a recurrence relation)

6) What are the three steps in an inductive proof?

- Prove for base case
- Assume true for n, prove for larger n

7) Use an inductive proof to show the sum of the first n integers is $\frac{n(n+1)}{2}$

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

$$\sum_{k=1}^{1} k = 1 = \frac{1(1+1)}{2}$$

$$1 = \frac{1(2)}{2}$$

$$1 = 1 \qquad \sum_{k=1}^{m} k = \frac{m(m+1)}{2}$$

$$\sum_{k=1}^{m+1} k = \frac{(m+1)(m+2)}{2}$$

$$\sum_{k=1}^{m} k + \sum_{m+1}^{m+1} = \frac{(m)(m+1)}{2} + (m+1)$$

$$= \frac{(m)(m+1)}{2} + (m+1)$$

Opening Questions

1. Define Big-O, Omega and Theta notation
2. In your own words explain what a recurrence relation is, what do we use recurrence relations for, why do we solve recurrence relations?

Recurrence Relation Solution Approach—Guess and prove by induction

Guess (or are given Hint) at form of solution, probe it is the solution

- Using definition of BIG-O or θ
- Using induction
  - Probe Base Case (if boundary condition given)
  - Assume true for some n
  - Prove true for a larger n

Example:

$$T(n) = 4T(n/2) + 2 \text{ guess } T(n) = O(n^3) \text{ ??}$$

Assume $T(k) \leq ck^3$ for some $k < n$, use assumption with $k = n/2$, then prove it for $k = n$

$T(n/2) \leq c(n/2)^3$ merge with recurrence

$$T(n) \leq 4c(n/2)^3 + n$$
$$T(n) \leq \frac{c}{2n^3} + n$$
$$T(n) \leq cn^3 - \left(\frac{c}{2n^3} - n\right)$$
$$\left(\frac{c}{2n^3} - n\right) > 0 \text{ if } c \geq 2 \text{ and } n > 1$$
$$T(n) \leq cn^3 - \text{(something positive)}$$
$$T(n) \leq cn^3$$
$$T(n) = O(n^3)$$

1. $$T(n) = 4T(n/2) + n^3 \text{ guess } T(n) = \Theta(n^3)??$$

2. $$T(n) = 4T(n/2) + n \text{ guess } T(n) = O(n^2)??$$

Recurrence Relation Solution Approach - Iteration Method (repeated substitution)

Convert the recurrence relation to summation using repeated substitution (Iterations)

- Keys to Iteration Method
  - # of times iterated to get $T(1)$
  - Find the pattern in terms and simplify to summation

EXAMPLE

$$
\begin{aligned}
T(n) &= 4T\left(\frac{n}{2}\right) + n \\
T(n) &= n + 4T\left(\frac{n}{2}\right) \\
&= n + 4\left(4T\left(\frac{n}{4}\right) + \frac{n}{2}\right) \\
&= n + 4\left(\frac{n}{2} + 4\left(\frac{n}{4} + 4T\left(\frac{n}{8}\right)\right)\right) \\
&= (n + 2n + 4n + 64)T\left(\frac{n}{8}\right) \\
T(n) &= n + 2n + 4n + \cdots + 4^{\lg_2(n)}T(1) \\
&= n \sum_{k=0}^{\lg(n-1)} 2^k + \theta(n^2) \\
&= n\left(\frac{2^{\lg(n)} - 1}{2 - 1}\right) \\
T(n) &= n^2 - n \\
&= O(n^2)
\end{aligned}
$$

3. $T(n) = 3T(n/3) + \lg n$ proof by iteration/repeated substitution

4. $$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

5. $$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + n^2$$

6. $$T(n) = T(n-1) + n$$

Recurrence Relation Solution Approach – Master Method

For solving recurrences of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

- Compare growth of $f(n)$ to $n^{\log_b(a)}$

Case 1)

$$f(n) \leq cn^{\log_b(a)} \; T(n) = \Theta(n^{\log_b(a)})$$

Case 2)

$$c_1 n^{\log_b(a)} \leq f(n) \leq c_2 n^{\log_b(a)} \; T(n) = \Theta(n^{\log_b(a)} \lg(n))$$

Case 3)

$$f(n) \geq cn^{\log_b(a)} \; T(n) = \Theta(f(n))$$

7. $$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

8. $$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

9. $$T(n) = 4T\left(\frac{n}{2}\right) + n^3$$

Opening Questions

1. Mergesort is $\Theta(O(n \lg n))$ runtime in best case, worst case and average case. How much memory is needed for Mergesort on input size n? $O(n)$

2. Mergesort does all the work of sorting items in the Merge function, after recursively splitting the collection down to the base case. Briefly explain the difference with Quicksort.All the work for Mergesort is being done after the recurrsive calls, where as Quicksort does the work before the recurrsive calls.

Quicksort

A recursive divide and conquer algorithm.

- base case: a list of length one element is sorted.
- Divide "Partition" array into 2 sub-arrays with small #'s in the beginning, large #'s in second and known index dividing themWhat defines small and large? The dividing index may not be the middle.
- Conquer - recursively sort each sub-array
- Combine - Nothing to do

1. Write pseudocode for Quicksort (similar to Mergesort).

Algorithm 6 Quicksort

```
1: function QSort(A, p, r)                    ▷ The initial call is QSort(A, 1, n)
2:     if p < r then
3:         q ← Partition(A, p, r)    ▷ Moves small to left and large to right, q is the seperator
4:         QSort(A, p, q-1)                   ▷ The value at A[q] is the pivot, q is the idnex
5:         QSort(A, q+1, r)
6:     end if
7: end function
```

Partition idea (you should be able to do this in place): pick the last element in the current array as the "pivot", the number used to decide large or small. Then make a single pass of the array to move the "small" numbers before the "large" numbers and keep the "large" numbers after the "small" numbers. Then put the "pivot" between the two subarrays and return the location of the pivot.

2. Write iterative pseudocode for Partition. How much memory is needed?

Algorithm 7 Partition: Worst case runtime $O(n)$

```
1: function PARTITION(A,p,r)
2:     pivot ← A[r]
3:     i ← p − 1
4:     for j=p to r-1 do
5:         if A[j] ≤ pivot then
6:             Swap A[i+1] with A[j]
7:             increment i
8:         end if
9:     end for
10:    Swap A[i+1] with A[r]
11:    return i+1
12: end function
```

Quicksort Runtime:

$$T(n) = O(n) + T(\text{small items}) + T(\text{large items})$$
$$T(1) = O(1) + (O \leftrightarrow n-1) + (n-1 \leftrightarrow O)$$

If All Items $<$ Pivot

$$\begin{aligned} T(n) &= O(n) + T(n-1) + T(0) \\ &= O(1) + T(1-1) + O(1) \\ &= O(1) + T(0) + O(1) \\ &= O(1) \end{aligned}$$

$$\begin{aligned} T(n) &= O(n) + T(n-1) \\ &= O(n) + O(n-1) + T(n-2) \\ &= O(n) + O(n-1) + O(n-2) + T(n-3) \\ &= O(n) + O(n-1) + O(n-2) + O(n-3) + \cdots + T(1) \\ &= O(n) + O(n-1) + O(n-2) + O(n-3) + \cdots + O(1) \\ &= O\left(n^2\right) \end{aligned}$$

If Pivot is Median

$$\begin{aligned} T(n) &= O(n) + 2T\left(\frac{n}{2}\right) \\ &\quad (\text{Master method: } a = 2, b = 2, f(n) = O(n)) \\ &= n^{\log_b(a)} \\ &= n^{\log_2(2)} \\ &= n^1 \\ &= O(n \lg n) \end{aligned}$$

What if pivot divides items $\frac{1}{10}$ on oneside and $\frac{9}{10}$ on other

$$\begin{aligned} T(n) &= T\left(\frac{n}{10}\right) + T\left(\frac{n}{10}\right) + O(n) \\ &\cdots \\ &= \log_{\frac{10}{9}}(n) \\ &= O(n \lg n) \end{aligned}$$

As long as the split is some sort of ratio, you get $O(n \lg n)$ as the runtime for quicksort. Only in the rare case of all the data being on one side of the partition that gives $O(n^2)$ as the partition. To fix this, pick 3 items randomly, chose the middle of those 3, swap that middle with the last index and you'll be guaranteed to have $O(n \ln n)$.

Algorithm 8 Partition Improved

```
1: function PARTITIONI(A,p,r)
2:     b, c, d are random indexes from A.
3:     Swap A[r] with MEDIAN(A[b], A[c], A[d])
4:     pivot ← A[r]
5:     i ← p − 1
6:     for j=p to r-1 do
7:         if A[j] ≤ pivot then
8:             Swap A[i+1] with A[j]
9:             increment i
10:        end if
11:    end for
12:    Swap A[i+1] with A[r]
13:    return i+1
14: end function
```

3. Demonstrate Partition on this array.

4. What do you think the best possible outcome would be for a call to Partition, and why? What about worst possible outcome?

5. Write (and solve) recurrence relations for Quicksort in the best case partition and worst case partition.

6. What is there is a pretty bad, but not awful, partition at every call. Try always a 9 to 1 split from partition. Write and solve recurrence relation.

With all the other sorts we could describe a particular input order that would yield worst case run time.

7. How can we avoid a particular input order yielding worst case run time for quicksort?

Visual Sorting Software By A. Alegoz, previous CS430 student (1.8Mb zipped, Win only)http://www.cs.iit.edu/~cs430/IITSort.zip

Opening Questions

1. Explain the difference between the Binary Search Tree Parent-Child value relationship and the Heap Property Parent-Child value relationship. BST: the right node is larger than the parent node and the left node is smaller than the parent node. In Heaps, the parent is either larger than both children (MaxHeap) or smaller than both children (MinHeap).

2. Binary search trees are a dynamic data structure that uses left-child and right-child pointers to represent the tree. How is this different from a heap? In a heap, you can take the index of a node, apply $\lfloor \frac{\text{node}}{2} \rfloor$ to get the parent node. So you can wall up or down a heap whereas a binary tree can only walk down.

Heaps

Since a heap is a nearly complete binary tree and will always grow and shrink the rightmost bottom leaf, you can implement a head with an array instead of needing pointers (as is needed for a binary search tree which can grow/shrink at any node. Example of a MaxHeap showing array implementation:

1. At what index position is the largest element in a MaxHeap? We have to know how to easily move around a Heap. Can you devise a formula to relate the index of a parent to the indexes of its children? How about a formula for the index of a child to the index of its parent? The largest element in a Max Heap is at index 1 (given 1-based indexing). The formula is index_parent $= \left\lfloor \frac{\text{child\_index}}{2} \right\rfloor$.

2. If a heap was one larger, where does the tree have to gain a node from when done? If a heap was one smaller, where does the three have to lose a node when done?

3. Considering your answer to #2, try to devise a way to ExtractMax from this maxheap. What is the runtime in terms of heapsize?

Algorithm 9 ExtractMax

```
1: function ExtractMax(heap)                          ▷ 1 based indexing, not 0
2:     Save the value at the root
3:     Move heap[heapsize] to heap[1]
4:     MaxHeapify(heap)
5: end function
```

Save the current value at the root. Then move 5 (which is at the heapsize index) to the root, and then heapify.

4. Considering your answer to #2, try to devise a way to Insert(20) into this maxheap. What is the runtime in terms of heapsize?

Both the above ExtractMax and Insert assumed we already had a heap. To efficiently build a heap, we put all the items to insert in an array. Call MaxHeapify (walk value down) from index $\frac{\text{heapsize}}{2}$ up to the root (index 1).

5. Write pseudocode for this BuildHeap algorithm and demonstrate on this data. What is the runtime in terms of n, the size of B? $B = [15\ 8\ 4\ 9\ 3\ 16]$ B is not a max heap.

Algorithm 10 BuildHeap

```
1: function BuildHeap(heap)
2:     MaxHeapify(heap, 1, heapsize/2)▷ Precondition: Both children of that index are valid max
   heaps
3:                                  ▷ Any index position larger than heapsize/2 is a leaf→ valid max heap
4: end function
```

Runtime: must call $MaxHeapify$: $\frac{heapsize}{2}$ times. $MaxHeapify$ in the worst case runs at $O(\lg heapsize) = O(\lg n)$. However, working at $heapsize$, at most $MaxHeapify$ works once (incase the only child is larger than the node).

$$\theta(n) = \lg(n) + 2\lg(n-1) + 4\lg(n-2) + \cdots + \left(\frac{n}{4}\right)(1) + \frac{n}{2}(0)$$
$$= \text{linear time.}$$

6. Write the loop invariant for BuildHeap and prove that it works.After the jth iteration (index of the item we call MaxHeapify on: (range $\frac{n}{2} \to 1$). Before call on MaxHeapify on jth iindex in the heap $\frac{n}{2} \to 1$, $\frac{n}{2} + 1 \to n$ are max heaps, which means child of j index are max heaps.
INIT: We know all index positions $> n/2$ are leafs or empty, therefore, they are already maxheaps.
MAINT: If $p(j) \to p(next\_j)$

7. How can we use a maxHeap and extractMax to sort? Build the heap in $O(n)$ time, then extractMax which saves the root, then sorts the heap. The initially saved root gets moved to another array into the index $heapsize - 1$. Then extract max is run again, and the saved value is moved into $heapsize - 2$. The runtime is $O(n \lg n)$.

Opening Questions

- If we have 17 distinct items to sort, what is the height of the decision tree that represents all possible orderings of the 17 items?

Comparison Sorts

The sorted order they determine is based only on comparisons between the input elements. That is, given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i = a_j$, or $a_i > a_j$ to determine their relative order.
Claim: We must make at least $\Omega(n \lg n)$ comparisons in the general case.
The decision-tree model

- Abstraction of any comparison sort.
- Represents comparisons made by a specific sorting algorithm on inputs of a given size.
- Abstracts away everything else: control and data movement
- View the tree as if the algorithm splits in two at each node, based on the information it has determined up to that point; The tree models all possible execution traces.

Each internal node is labeled by indices of array elements **from their original positions**. Each leaf is labeled by the permutation of orders that the algorithm determines.

1. How many possible permutations of n items are there (at least one of them must be sorted)? In the decision tree model, how many leaves are there for n items? For a binary tree to have that many leaves, how many levels does it need? Why do we care about how many levels are in the decision tree?

Linear Time Sorting – Counting Sort

If we do not use comparison of keys to sort data, how can we sort data? It seems comparing items is necessary for sorting.

- Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When $k = O(n)$, the sort runs in $T(n)$ time (best if $k \lll n$)
- For each input element x, count how many elements are equal to x then use this to count how many elements are $\leq x$. This information can be used to place element x directly into its position in the output array.
- Does not sort in place, needs 2^{nd} array size n and 3^{rd} array size k

2. Write pseudocode for counting sort.

Algorithm 11 Counting Sort

```
1: function COUNTING SORT
2:
3: end function
```

3. Demo counting sort on this data:

2	5	3	0	2	3	0	3

Counting sort is best on large amounts of data where the range of the data is small. Like many single digit numbers. However, we can use counting sort multiple times to sort multiple digit numbers, starting with the right most digit, etc. This is called Radix Sort.

Table 2: Radix Sort

329		72 0		7 2 0		3 29
457		35 5		3 2 9		3 55
657		43 6		4 3 6		4 36
839	→	45 7	→	8 3 9	→	4 57
436		65 7		3 5 5		6 57
720		32 9		4 5 7		7 20
355		83 9		6 5 7		8 39

An important property of counting sort is that it is **stable**: numbers with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two numbers by the rule that whichever number appears first in the input array appears first in the output array. Normally, the property of stability is important only when satellite data are carried around with the element being sorted. Counting sort's ability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

4. Which of these sorts can be stable:

    - Insertion Sort
    - Selection Sort
    - Bubble Sort
    - Merge Sort
    - Quick Sort
    - Heap Sort