CPS222 Lecture: Algorithm Analysis

last revised January 10, 2013

Objectives:

1. To introduce the notion of algorithm analysis in terms of time and space (by counting instructions/memory cells)
2. To introduce the O(), omega(), theta() and omicron() measures of complexity.
3. To introduce the functions commonly encountered when analyzing algorithms.
4. To show how the O() measure can be obtained by inspecting an algorithm.
5. To explain the significance of the asymptotic complexity of an algorithm.

Materials:

1. Projectable of detailed analysis of bubble sort
2. Projectable of rates of growth table and graph
3. Projectables of various solutions to the array subsequence problem
4. Demo program of various solutions to the above

I. Introduction to algorithm analysis

A. One of the things one discovers is that there are often several ways of doing the same job on a computer.

Example: For each type of Java collection, we can choose between two different implementations. One mark of maturity as a Computer Scientist is the ability to choose intelligently from among alternative ways of solving a problem. This implies some ability to measure various options to assess their "cost". The two most common measures are:

1. Time

    a) CPU cycles (typically on the order of 1 ns [order of magnitude] - this number declined dramatically during much of the history of computing but has remained relatively constant for the last 10 years or so - note that when I revised a 2004 lecture I didn't actually have to change this number! (It's now about 0.3-0.5 ns per cycle - same order of magnitude)

    b) Disk accesses (typically ~ 10 ms [order of magnitude again] - this number hasn't changed much for decades.)

    Note - btw - the $1 : 10^7$ ratio between the above!

2. Space

    a) Main memory and cache memory (typically measured in GB and MB or even KB respectively)

    b) Secondary storage (typically measured in blocks of a few KB)

3. Also to be considered is programmer effort to write and maintain the code, of course.

B. Often, there is a trade-off between time and space; one can gain speed at the expense of more space and vice versa. (But some bad algorithms are hogs at both)

C. One must also consider the types of operations to be performed.

Example - we have already noted that, in the case to the two standard Java implementations of the List collection, one (the ArrayList) is much more efficient when accessing items by position in the middle of the list, while the other is much more efficient when doing operations near either end of the list, like inserting a new first element.

D. Therefore, one must often analyze data structures and/or algorithms for performing various tasks in order to discover the best method. Such analyses are generally done with a view to measuring time or space as a function of "N" - some parameter measuring the size of a particular instance of the problem (e.g. the number of items in the list.)

1. If N is sufficiently small, then it may be that any choice is fine, and the one that is easiest to implement correctly may be the best.

2. As N grows large, there can be huge differences in performance between different solution strategies.

   Example: The RSA encryption/decryption algorithm requires raising a message block to the power of either the public or private key (e or d) as appropriate. This could entail a very large power.

   a) Consider the case of raising a number to a power that itself has 200 digits - i.e. calculating $block^{power}$, where power is a 200 digit number. The naive way to do calculate this would be to multiply block by itself power times, which would take on the order of 10^{200} multiplications. If we could do a multiplication in 1 ns (unrealistically fast even for modern computers for a large number), this would take 10^{191} seconds or more than 10^{188} hours or 10^{186} days or 10^{183} years!

   b) However, there is a much faster algorithm that can do the job with no more than about 1300 multiplications - needing a small fraction of a second!

      We mentioned this in CPS221. Can anyone recall the basic strategy?

      ASK

      The key is that we just calculate P^1, P^2, P^4, P^8, P^{16}, P^{32} ... - and then form P^n as a product of the terms corresponding to 1's in the binary representation of n - e.g. if n is 1025 (10000000001 in binary), then we would calculate P^{1025} as $P^{1024} * P^1$, using just 10 multiplications

to create the initial powers and then one multiplication to calculate the actual result.

3. Therefore, we are particularly interested in analyzing how the time or space required by a given solution strategy GROWS as the size of the problem grows. We call this the AYSMPTOTIC COMPLEXITY.

E. While analysis of both time and space is possible, most of the the time what we focus on is the time complexity of an algorithm, and this is what we will focus on in this lecture. The same principles can easily be applied to space complexity as well.

F. An example: Consider the simplest form of bubble sort. Assume our goal is to sort an array of n numbers, which are stored in an array x[0] .. x[n-1].

PROJECT

```
for (int i= 0; i < n-1; i ++)
    for (int j = 0; j < n-1; j ++)
     if (x[j] > x[j+1])
         // Swap x[j] with x[j+1];
```

1. Suppose it takes some time t1 to set up a for loop; t2 to increment the loop variable and test it against the limit; t3 to compare two elements, and t4 to switch them. Suppose, further, that the probabilityof two elements being out of order and needing to be switched is p (where p can be as small as 0 if the array is already sorted and as large as 1.0 in the worst case) (Presumably, each of the "t" constants is of comparable magnitude -probably on the order of < 10 ns with modern computers)

2. Then, since we go through the outer loop n-1 times and we go through the inner loop n-1 times for each time through the outer loop, it is not hard to see that our overall time is

t1 + (n-1) (t2 + t1 + (n-1) (t2 + t3 + pt4))
= (t2 + t3 + pt4) * n^2 + (t1 - t2 - 2t3 - 2pt4) * n + (t3 + pt4)

This has the form: $c1 * n^2 + c2 * n + c3$

where $c1 = t2 + t3 + pt4$; $c2 = t1 - t2 - 2t3 - 2pt4$, $c3 = t3 + pt4$

(observe that c1 and c3 are necessarily positive, assuming all the times are. c2 might be negative.)

3. For large n, the last two terms become arbitrarily small when compared to the first term. Thus, as n gets large, the run time grows with the square of n - i.e. doubling n will roughly quadruple the run time; tripling n will multiply the run time by 9, etc.

4. Further, we note that while the "t" values are basically determined by the particular hardware on which the problem is run, the fact that the execution time grows proportionally to n^2 is a fundamental property of the algorithm, regardless of hardware. Therefore, we say that the bubble sort is an $O(n^2)$ algorithm. (We will formally define what we mean by "big O" shortly.) This statement is as valid today as it was when bubble sort was first analyzed in the 1950's!

5. In particular, if sorting 100 items takes (say) 1ms of CPU time on a certain CPU, then we expect:

   a) 200 items to take about

      ASK

      4ms

   b) 1000 items to take about

      ASK

      100 ms

c) 10,000 items to take about

ASK

10,000 ms = 10 seconds

6. In this examples, we have done a rather detailed analysis of a simple algorithm. If we had to do this every time, analysis could be very difficult. However, we will soon see that we can arrive at an order of magnitude estimate - a "big O" rating - fairly easily.

II. Definitions of Big-O and Related Measures

A. Formally, we say that a function T(n) is O(f(n)) if there exist positive constants c and n_0 such that:

T(n) <= c * f(n) whenever n >= n_0.

1. We then say that T(n) = O(f(n)) - note that the less precise O function appears on the right hand side of the equality.

2. In the case of the bubble sort, we saw that T(n) = c1 n^2 + c2n + c3 and claimed that this is O(n^2).

   To show that this claim holds, let n_0 be 1 and let c be c1 + max(c2, 0) + c3.

   Then we have c * f(n) = (c1 + max(c2, 0) + c3) * n^2

   = c1 * n^2 + max(c2, 0) * n^2 + c3 * n^2

   Clearly c1 * n^2 + c2 * n + c3 is <= this whenever n >= 1.

B. The O() measure of a function's complexity gives us an upper bound on its rate of growth.

1. As such, it is not necessarily a tight bound. For example, if we know that a particular function happens to be O(n), we are allowed by the definition of Big-O to say it is $O(n^2)$ or (n^3) or even $O(2^2)$, since n is less than or equal to each of these other functions whenever n >= 1.

   Of course it would be silly to do so - but the point is that big O only tells us that the behavior can be no worse than some bound.

2. For this reason, we sometimes use one or more additional metrics to characterize the behavior of an algorithm:

   a) Ω (Big Omega) measures the LOWER bound:

      (1) We say that f(n) is Ω(g(n)) if there exist positive constants c and n_0 such that

      $f(n) >= c * g(n)$ whenever $n >= n_0$.

      (Some writers say "for infinitely many values of n" instead).

      (2) Alternately, we can say that f(n) is Ω(g(n)) iff g(n) is O(f(n))

      (3) Sometimes we speak of the Ω complexity of a <u>problem</u>, by which we mean that any algorithm for solving that problem must have at least this complexity.

      Example: we will show later in the course that sorting based on comparison of items is Ω(n log n) - i.e. any algorithm for doing this must have complexity at least n log n.

      (4) Note that - as with big O, a big Omega bound is not necessarily tight - e.g. we can say that ANY non-zero function is Omega(1).

b) Θ (Theta) provides a TIGHT bound.

(1) We say that f(n) is $\Theta(g(n))$ if there exist positive constants c1 and c2 and n_0 such that

$c1 * g(n) <= f(n) <= c2 * g(n)$ whenever $n >= n_0$

(2) Alternately, we can say that f(n) is Θ (g(n)) if is true both that $f(n) = O(g(n))$ and $f(n) = \Omega(h(n))$ or that $f(n) = O(g(n))$ and $g(n) = O(f(n))$

c) o (omicron - Little oh) provides a STRICTLY GREATER UPPER BOUND.

We say that f(n) is $o(g(n))$ if f(n) is $O(g(n))$ but not $\Theta(g(n))$.

C. An example: Consider the function $f(n) = 2*n^3 + 11$. We can show that:

f(n) is $O(n^3)$, $O(n^{\wedge 4})$, $\Omega(n)$, $\Omega(n^3)$, and $\Theta(n^3)$, but it is NOT $o(n^3)$, though it is $o(n^4)$.

1. f(n) is $O(n^3)$

How can we show this? ASK

To show that $2* n^3 + 11$ is $O(n^3)$, we need to find positive constants c and n0 such that $2* n^3 + 11 <= c* n^3$ whenever $n >= n_0$

There are many values of c and n_0 that would work - one possibility is $c = 3, n_0 = 3$

Is $2* n^3 + 11 <= 3 * n^3$ whenever $n >= 3$?

Equivalent to asking is $11 < n^3$ whenever $n >= 3$, for which the answer is clearly yes.

2. f(n) is $O(n^4)$

How can we show this? ASK

We can use the same c and n_0 as above.

3. f(n) is $\Omega(n)$

   How can we show this? ASK

   To show that 2*n^3 + 11 is $\Omega(n)$, we must find positive constants c and n_0 such that $2* n^3 + 11 >= c * n$ for all $n >= n_0$

   This one is easy: c = 1 and n0 = 0 works!

   Is $2* n^3 + 11 >= n$ whenever $n >= 0$? Obviously, yes

4. f(n) is $\Omega(n^3)$

   How can we show this? ASK

   To show that $2* n^3 + 11$ is $\Omega(n^3)$, we must find positive constants c and n_0 such that $2* n^3 + 11 >= c* n^3$ whenever $n >= n_0$

   We can use c = 2 and $n_0 = 0$, since

   $2* n^3 + 11 >= 2* n^3$ for all n!

5. f(n) is $\Theta(n^3)$.

   How can we show this? ASK

   a) Two approaches. One approach is to observe that it is both $O(n^3)$ and $\Omega(n^3)$, which makes it $\Theta(n^3)$ by the definition of Θ

   b) Or, we could show this directly by finding positive constants c1, c2 and n_0 such that

      $c1* n^3 <= 2* n^3 + 11 <= c2* n^3$ whenever $n >= n_0$

      One set of values that works is c1 = 2, c2 = 3, $n_0 = 3$ - then we have

      $2* n^3 <= 2* n^3 + 11 <= 3* n^3$ whenever $n >= 3$ - which is true

6. f(n) is NOT l o(n^3)

   How can we show this? ASK

   We have shown that f(n) is O(n^3), but also that it is $\Theta(n^3)$; therefore it is not little-oh (n^3).

7. f(n)is o(n^4)

   How can we show this? ASK

   We have shown that f(n)is O(n^4); we must now show it is NOT $\Theta(n^4)$, which requires that we show it is NOT $\Omega(n^4)$ since we know it is O(n^4)

   a) Suppose $2* n^3 + 11$ were $\Omega(n^4)$. Then there would have to exist positive constants c and n_0 such that $2* n^3 + 11 >= c* n^4$ whenever $n >= n_0$

   b) But no such constants exist - if they did, we would have

      $\frac{2* n^3 + 11}{c* n^4} >= 1$ for all $n >= n_0$, which is equivalent to

      $\frac{2}{c*n} + \frac{11}{c*n^4} >= 1$ whenever $n >= n_0$

      But since both terms approach 0 as n grows arbitrarily large, this cannot be so.

D. Another way of getting at the relative size of two functions is what happens to their RATIO as n approaches infinity.

1. If lim n -> infinity f(n)/g(n) = 0, then f(n) is little o(g(n))
2. If lim n -> infinity f(n)/g(n) = some nonzero constant, then f(n) is Θ(g(n))

3. If lim n -> infinity f(n)/g(n) is unbounded, then f(n) is Ω(g(n))

III. **Functions that Commonly Show up in Algorithm Analysis**

A. As the book noted, there are seven functions of n that frequently show up when analyzing algorithms. That's not to say that other functions don't show up - only that these seven are the most common.

What are they?

ASK

1. Linear Complexity - O(1)

   a) We say than an operation has O(1) complexity if the time it requires is the same regardless of the size of the problem.

   b) Often, the individual steps in an algorithm have O(1) complexity. For example, algorithms like sorting often involve an operation like the comparison of two items for relative order. An individual comparison is normally O(1) - the complexity of the overall algorithm comes from the number of such comparisons it requires.

   c) Occasionally, a full algorithm will have O(1) complexity - though this is uncommon. You may recall, for example, that operations on a hash structure (such as a Java `HashSet` or `HashMap`) have O(1) average class complexity.

2. Logarithmic Complexity - O(log n)

   a) Many algorithms proceed by a divide and conquer process. Often, the problem is solved by dividing the original problem into two subproblems, with the solution to the overall problem being a composition of the solutions to the subproblems. Each of the subproblems, in turn, is solved by dividing it in two ... and this process continues until the problem becomes trivial (often of size <= 1).

   b) Any problem that is solved this way requires O(log n) divisions into subproblems, with the overall complexity determined by the complexity of composing the solution to the original problem from the subproblems.

   c) As a simple example, consider binary search in an ordered list.

      (1) Look at the middle item. If the desired item matches this, the search is done. If it is less than the middle, continue searching in the first half of the list and ignore the last half; it is greater, continue with the last half and ignore the first.

      (2) The size of the problem goes n, n/2, n/4 ... - and becomes trivial when there is just one item left, since it is either what we want or the desired item does not exist.

      (3) The number of divisions is, of course, O(log n)

3. Linear Complexity - O(n)

   a) If the number of operations in an algorithm is directly proportional to the size of the problem, we say it is O(n).

   b) A simple example: searching for an item in a structure like an array that is not ordered. The only way to perform the search is to look at items - generally starting with the first - until the

desired item is found or all the items have been looked at and we conclude the desired item is not present.

(1) In the best case, we find the item on the first try.

(2) In the worst case, we have to look at all n items because the item is the last one we look at, or is not present at all - so this has O(n) worst case complexity.

(3) On average, we look at half the items (n/2). We say that this, too, is O(n) [c is 0.5 in our definition of Big-O].

4. O(n log n)

   a) Often, divide and conquer algorithms entail significant effort to compose the overall solution from the solutions to the subproblems.
   b) We will see a number of algorithms that require O(log n) divisions of the original problem, but require O(n) total effort after each division to compose the solutions to the subproblems.
   c) Such algorithms have complexity O(n * log n) = O(n log n)

5. Quadratic Complexity - $O(n^2)$

   a) We will meet quite a few algorithms which take time proportional to the square of the size of the problem.

   b) We saw one such algorithm earlier in this lecture. As you recall, bubble sort (and actually quite a few other sorting algorithms) has complexity $O(n^2)$.

6. Cubic and other Polynomial Complexities - $O(n^3)$ .. $O(n^k)$ where k is a constant.

   a) We will meet a few algorithms whose effort grows as the cube of the size of the problem.

b) Occasionally, we will encounter other polynomials like $O(n^4)$

7. Exponential Complexity - $O(2^n)$

   a) Consider a problem such as listing all the possible subcommittees that could be formed from a group of n people.

      (1) There are n possible subcommittees of one member.

      (2) There are n * (n-1) / 2 possible subcommittees of two members.

      (3) There are n * (n-1) * (n-2) / 6 possible subcommittees of three members
      ...
      (4) There is one possible subcommittee of the whole (all n members).

      (5) You could find out the total number of possibilities by adding all these up, or a quicker route would be to observe that there are two possibilities for each member - on the committee or not on the committee - each of which yields a different subcommittee.

      (6) The total is therefore 2^n - which includes the possibility of a subcommittee with no members.

   b) An algorithm like this is said to have exponential complexity. Note that a characteristic of such complexity is that it doubles when the size of the problem increases by 1 - which makes exponential algorithms practical only for small problems.

      Example: An exponential algorithm applied to a problem of size 20 would require a million steps, 30 a billion steps ...

B. It is helpful to have some understanding of how these functions compare to one another

1. Values of the various functions for different values of NA

   PROJECT

2. Graph of the above

   PROJECT

C. If we know the time needed by a particular algorithm for one value of N, we can project is time for other, higher values of n

1. An $O(n^2)$ algorithm - if it needs 100 for 1000 items, it will need 400 ms for 2000, 900 ms for 3000, 1.6 sec for 4000 etc.

2. An O(n log n) algorithm - it it needs 100 ms for 1000 items, it will need (2000 log 2000)/(1000 log 1000)*100 = 2*1.1*100 = 220 ms for 2000, (3000 log 3000)/(1000 log 1000)*100 = 3*1.15*100 = 345 ms for 3000, (4000 log 4000)/(1000 log 1000)*100 = 4*1.2*100 = 460 ms for 4000 etc.

D. Observe: if an O(n log n) algorithm was 10 times slower than an $O(n^2)$ algorithm for n = 1, it would beat the $O(n^2)$ algorithm for all n > 60 or so. If it were a 100 times slower, it would beat the $O(n^2)$ algorithm for all n > 1000 or so.

E. One important point worth noting is that complexities all into two broad categories: POLYNOMIAL COMPLEXITY and EXPONENTIAL COMPLEXITY.

1. We say T(n) has polynomial complexity if for some integer constant k it is true that

   T(n) is $O(n^k)$

2. We say that T(n) has exponential complexity if

   T(n) is omega(2^n)

3. A practical ramification of this distinction is that problems that have polynomial complexity solutions are potentially solvable algorithmically (if not now, then with increasing computing speeds). Problems for which the best algorithm has exponential complexity will NEVER be algorithmically solvable for even moderate size no matter how much computing speed may increase, because increasing the size of the problem by 1 at least doubles the run time!

IV.Computing Complexities

A. To compute the order of a time or space complexity function, we use the following rules:

1. If some function T(n) is a constant independent of n (T(n) = c), then T(n) = O(1).

2. We say that O(f(n)) is less than O(g(n)) if for any c >= 1 we can find an n_0 such that g(n)/f(n) > c for all n > n_0.

   a) In particular, we observe the following relationship among functions frequently occurring in analysis of algorithms:

      $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

   b) Example: we way that O(n) < O(n log n) because the ratio n log n / n = log n can be made greater than any desired value c by choose n_0 such that log n_0 > c.

3. Rule of sums: If a program consists of two sequential steps with time complexity f(n) and g(n), then the overall complexity is O(max(f(n),g(n))). That is, O(f(n)) + O(g(n)) = O(max(f(n),g(n))). Note that if f(n) >= g(n) for all n >= n0 then this reduces to O(f(n)).

   Corollary: O(f(n)+f(n)) = O(f(n)) - NOT O(2f(n))

4. Rule of products: If a program consists of a step with complexity g(n) that is performed f(n) times [i.e. it is embedded in a loop], then the overall complexity is O(f(n)*g(n)), which is equivalent to O(f(n)) * O(g(n))

   Corollary: O(c*f(n)) = O(f(n)) since O(c) = 1

5. Example - quicker analysis of bubble sort

```
for (int i= 0; i < n-1; i ++)
    for (int j = 0; j < n-1; j ++)
        if (x[j] > x[j+1])
            // Swap x[j] with x[j+1];
```

   PROJECT CODE AGAIN

   a) The comparison (and possible exchange) step has complexity 1 but is embedded in a loop [for (j=0; j < n-1; j++)] that has complexity O(n).

   b) The inner loop as a whole consists of a setup step O(1) and overhead O(n). Therefore, the time complexity of the inner loop is O(1)+O(n)+O(n) = O(n).

   c) The outer loop consists of a setup step O(1) and overhead O(n) + the inner loop which has O(n) complexity done O(n) times and hence is $O(n^2)$.

   d) Therefore, the time for the outer loop - and the overall program - is O(1) + O(n) + O(n^2) = $O(n^2)$.

B. It is often useful to calculate two separate time or space complexity measures for a given algorithm - one for the average case and one for the worst case. For example, some sorting methods are O(nlogn) in the average case but $O(n^2)$ for certain pathological input data.

C. While these measures of an algorithm describe the way that its time or space utilization grows with problem size, it is not necessarily the case that if f1(n) < f2(n) then an algorithm that is O(f1(n)) is better than one that is O(f2(n)). If it is known ahead of time that the problem is of limited size (e.g. searching a list that will never contain more than ten items), then the algorithm with worse behavior for large size may actually be better because it is simpler and thus has a smaller constant of proportionality.

D. Also, in many practical situations one must also consider programmer time and effort to create and maintain an algorithm. In some cases, a "poorer" algorithm may be preferred - especially for code to be run only once or infrequently.

V. **An Extended Example**

The following is an example of a problem where efficiency of the algorithm makes a big difference (taken from Programming Pearls article by Jon Bentley in 9/84 CACM - has also been quoted in various texts):

A. Consider the following task: given an array x[0..N-1] of numbers, find the maximum sum in any CONTIGUOUS subvector - e.g. if x is the array

31 -41 59 26 -53 58 97 -93 -23 84

the maximum sum is x[2] + x[3] + x[4] + x[5] + x[6] = 187

B. Observe:

1. If all the numbers are positive, the task is trivial: take all of them.

2. If all the numbers are negative, the task is also trivial: take a subvector of length 0, whose sum, therefore, is 0.

3. The problem is interesting if x includes mixed signs - we include a negative number in the sum iff it lets us "get at" one or more positive numbers that offset it.

   a) In the above, we included -53 because 59 + 27 on the one side or 58 + 97 on the other more than offset it. The contiguous requirement would force us to omit one or the other of these subvectors if we omitted -53.

   b) We did not include -41. It would let us get at 31, but that is not enough to offset it. Likewise, we did not include -93.

C. We will consider and analyze four solutions:

1. The most immediately obvious - but poorest - method is to form all possible sums:

   PROJECT

   a) Time complexity?

      ASK

      The outer for is done n times. Each time through the outer for the middle for is done 1 to n times, depending on l. (The average is n/2 times.) The inner for is done 1 to n times each time through the middle for, depending on l and u. (The average is n/3 times.) Thus, the sum := sum + x[i] statement is done:

      n * (n/2) * (n/2) = $n^3/6$ = O(n^3/) times

   b) Implication: doubling the size of the vector would increase the run time by a factor of 8.

      DEMO: Run demo program for n = 500, 1000, 2000

After initial run, ask class to predict time for next

Write times on board

2. A better method is to take advantage of previous work, as follows:

PROJECT

a) Complexity? - ASK

The outer for is done n times; the inner for 1..n for each time hrough the outer (average n/2). The inner begin..end, then, s done:

$n * (n/2) = n^2/2 = O(n^2)$ times.

This is much better.

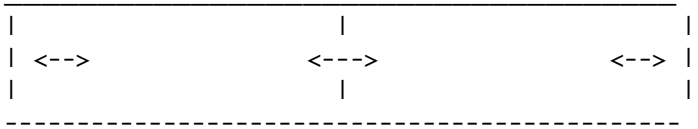
b) DEMO: Run demo program for N = 2000, 10,000, 50,000

Again ask class to predict running times after first, and record on board

3. An even better method is based on divide and conquer:

a) Divide the array in half. The best sum will either be:

(1) The best sum of the left half
(2) The best sum of the right half
(3) The best sum that spans the division



b) We can find the best sum of each half recursively.

There are two trivial cases in the recursion:

(1) A subvector of length 0 has best sum 0.

(2) A subvector of length 1 has best sum either equal to its one element (if that element is positive) or 0 (if the element is negative.)

c) The best spanning sum can be found by finding the best sum to the left and to the right from midpoint and then adding

PROJECT, Go over Code

d) Analysis: Each non-trivial call to MaxSumRec involves an O(right - left + 1) loop + O(right - left +1) calls, each of which faces a problem of half the size.

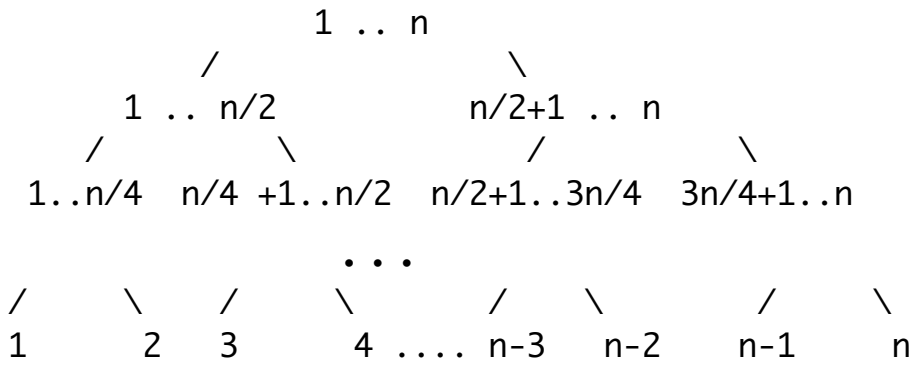
(1) The time complexity may be analyzed in terms of a recurrence equation. Let T(n) = the time to solve a problem of size n. Then we have:

T(1) = O(1)
T(n) [for n > 1] = 2T(n/2) + O(n)

(2) It can be shown mathematically that this recurrence has the solution:

T(n) = O(n log n) for all n > 1

(a) This can be seen intuitively from the following tree structure:



(we start with the problem of finding a solution in the vector x[1] .. x[n], which leads us to two subproblems for x[1] .. x[n/2], x[n/2 + 1] .. x[n], each of which leads to two subproblems ... Expansion of the tree stops when we reach n subproblems, each of size 1.)

(b) At each level, the total work is O(n)

(c) The number of levels is O(log N)

(d) Thus, the task done this way is O(n log N)

e) DEMO: Run demo program for N = 20,000; 200,000; 2,000,000

Again ask class to predict running times after first, and record on board

4. The best solution, however, beats even this. We use the following method:

a) Suppose that, in solving the problem for the vector x[1] .. x[n], I first obtain the solution for the vector x[1] .. x[n-1]. Then clearly, the solution for x[1] .. x[n] is one of the following:

(1) The same as the solution for x[1] .. x[n-1]

(2) or - A solution which includes x[n] as its last element. This latter solution consists of the sum of the best subvector ending at x[n-1] (which may be the empty vector) + x[n].

b) These observations lead to the following algorithm:

PROJECT Code

c) Clearly, this solution is O(n). Further, we cannot hope to improve upon O(n), since any algorithm must at least look at each element of the vector once, and thus must be at least O(n).

d) DEMO: Run demo program for N =1,000,000; 100,000,000

Again ask class to predict running times after first, and record on board