The algorithms portion of the course will require you to evaluate mathematical expressions. Recursive algorithms have a natural synergy with recurrence relations and hence for this course, you will be expected to solve simple recurrence relations. Find the upper-asymptotic bound to the recurrence relations below:

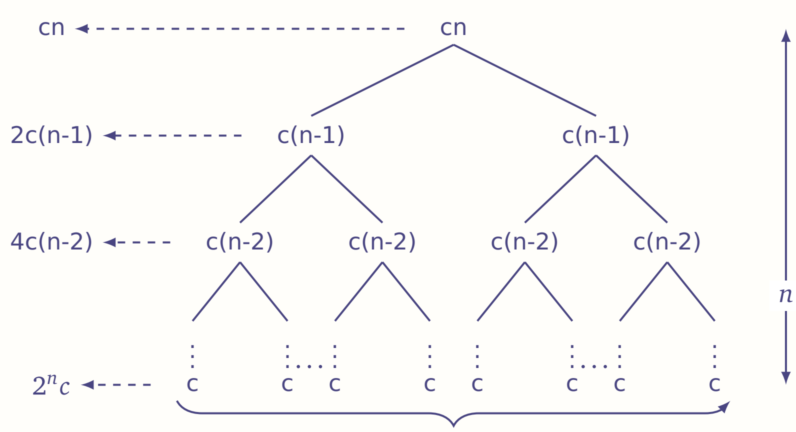
Solution: The following recurrence relations can be analyzed intuitively. In the case of simple recurrence relations, a potential strategy is to draw out the recurrence tree and find:

- the number of levels
- the work done at each level

and multiplying the two together to find the total work. Using this approach, there are three possibilities (corresponding to the three equations given!). ■

1. $T(n) = 2T(n-1) + cn$

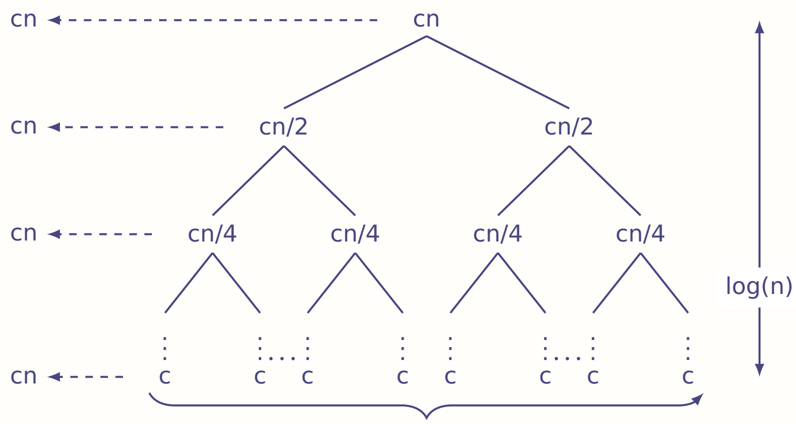
Solution: We can begin by visualizing the recursion tree:



The work are every level of the recursion tree is increasing and hence, the total work done is dominated by the "leafs". Intuitively we conclude that the asymptotic bound of the function is equal to the work done at the leaves and so: $T(n) = 2T(n-1) + cn = O(2^n)$ ■

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

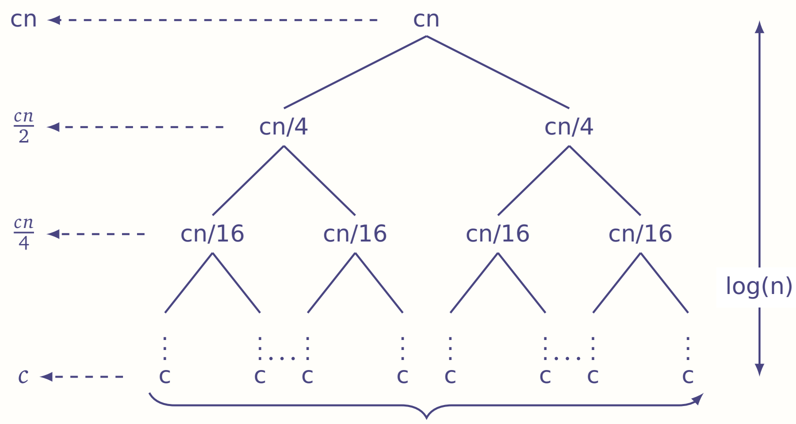
Solution: Doing the recursion tree out for this case, we notice that the amount of work is constant at every level:



Hence, the total amount of work done is equal to the amount of work at each level times the number of levels: $T(n) = 2T(n/2) + cn = O(n \log n)$ ■

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

Solution: Finally let's consider the case where the amount of work is deceasing at every level:



Like the first case, the total workload is dominated by one end of the tree (in this case the root). Hence, the total workload is dominated by the workload at the root: $T(n) = 2T(n/4) + cn = O(n)$. ■

Recurrence relations are a fascinating and area od discrete mathematics. Feel free to explore more advanced equations and solving techniques![1]

[1] http://discrete.openmathbooks.org/dmoi2/sec_recurrence.html

Here are several problems that are easy to solve in $O(n)$ time, essentially by brute force. Your task is to design algorithms for these problems that are significantly faster using binary search related ideas.

1. Suppose we are given an array $A[1..n]$ of n distinct integers, which could be positive, negative, or zero, sorted in increasing order so that $A[1] < A[2] < \cdots < A[n]$.

   (a) Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

Solution: Suppose we define a second array $B[1..n]$ by setting $B[i] = A[i] - i$ for all i. For every index i we have

$$B[i] = A[i] - i \le (A[i+1]-1) - i = A[i+1] - (i+1) = B[i+1],$$

so this new array is sorted in increasing order. Clearly, $A[i] = i$ if and only if $B[i] = 0$. So we can find an index i such that $A[i] = i$ by performing a binary search in B. We don't actually need to compute B in advance; instead, whenever the binary search needs to access some value $B[i]$, we can just compute $A[i] - i$ on the fly instead!

Here are two formulations of the resulting algorithm, first recursive (keeping the array A as a global variable), and second iterative.

```
⟨⟨Return any index i such that ℓ ≤ i ≤ r and A[i] = i⟩⟩
FindMatch(ℓ, r):
  if ℓ > r
      return None
  mid ← (ℓ + r)/2
  if A[mid] = mid                    ⟨⟨B[mid] = 0⟩⟩
      return mid
  else if A[mid] < mid               ⟨⟨B[mid] < 0⟩⟩
      return FindMatch(mid + 1, r)
  else                               ⟨⟨B[mid] > 0⟩⟩
      return FindMatch(ℓ, mid − 1)
```

```
FindMatch(A[1..n]):
  hi ← n
  lo ← 1
  while lo ≤ hi
      mid ← (lo + hi)/2
      if A[mid] = mid            ⟨⟨B[mid] = 0⟩⟩
          return mid
      else if A[mid] < mid       ⟨⟨B[mid] < 0⟩⟩
          lo ← mid + 1
      else                       ⟨⟨B[mid] > 0⟩⟩
          hi ← mid − 1
  return None
```

■

(b) Formulate a recurrence relation that describes your algorithm.

Solution: From the recursive formulation, we see that a constant ($O(1)$) amount of work is done at every step. Additionally every step tosses out half the array and so the algorithm is described by:

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

In both formulations, the algorithm ***is*** binary search, so it runs in $O(\log n)$ time. ■

(c) Suppose we know in advance that $A[1] > 0$. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. *[Hint: This is **really** easy.]*

Solution: The following algorithm solves this problem in $O(1)$ time:

```
FindMatchPos(A[1..n]):
  if A[1] = 1
      return 1
  else
      return None
```

Again, the array $B[1..n]$ defined by setting $B[i] = A[i] - i$ is sorted in increasing order. It follows that if $A[1] > 1$ (that is, $B[1] > 0$), then $A[i] > i$ (that is, $B[i] > 0$) for every index i. $A[1]$ cannot be less than 1. ■

2. Suppose we are given an array $A[1\,..\,n]$ such that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a ***local minimum*** if both $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are exactly six local minima in the following array:

9	7	7	2	1	3	7	5	4	7	3	3	4	8	6	9

Describe and analyze a fast algorithm that returns the index of one local minimum. For example, given the array above, your algorithm could return the integer 9, because $A[9]$ is a local minimum. *[Hint: With the given boundary conditions, any array **must** contain at least one local minimum. Why?]*

Solution: The following algorithm solves this problem in $O(\log n)$ time:

<u>LocalMin($A[1\,..\,n]$) :</u>
  if $n < 100$
    find the smallest element in A by brute force
  $m \leftarrow \lfloor n/2 \rfloor$
  if $A[m] < A[m+1]$
    return LocalMin($A[1\,..\,m+1]$)
  else
    return LocalMin($A[m\,..\,n]$)

If n is less than 100, then a brute-force search runs in $O(1)$ time. There's nothing special about 100 here; any other constant will do.

Otherwise, if $A[n/2] < A[n/2+1]$, the subarray $A[1\,..\,n/2+1]$ satisfies the precise boundary conditions of the original problem, so the recursion fairy will find local minimum inside that subarray.

Finally, if $A[n/2] > A[n/2+1]$, the subarray $A[n/2\,..\,n]$ satisfies the precise boundary conditions of the original problem, so the recursion fairy will find local minimum inside that subarray.

The running time satisfies the recurrence $T(n) \leq T(\lceil n/2 \rceil + 1) + O(1)$. Except for the $+1$ and the ceiling in the recursive argument, which we can ignore, this is the binary search recurrence, whose Solution is $T(n) = O(\log n)$.

Alternatively, we can observe that $\lceil n/2 \rceil + 1 < 2n/3$ when $n \geq 100$, and therefore $T(n) \leq T(2n/3) + O(1)$, which implies $T(n) = O(\log_{3/2} n) = O(\log n)$. ■

3. Suppose you are given two sorted arrays $A[1..n]$ and $B[1..n]$ containing distinct integers. Describe a fast algorithm to find the median (meaning the nth smallest element) of the union $A \cup B$. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1..8] = [2, 4, 5, 8, 17, 19, 21, 23]$$

your algorithm should return the integer 9. *[Hint: What can you learn by comparing one element of A with one element of B?]*

Solution: The following algorithm solves this problem in $O(\log n)$ time:

```
MEDIAN(A[1..n], B[1..n]):
  if n < 10^100
      use brute force
  else if A[n/2] > B[n/2]
      return MEDIAN(A[1..n/2], B[n/2+1..n])
  else
      return MEDIAN(A[n/2+1..n], B[1..n/2])
```

Suppose $A[n/2] > B[n/2]$. Then $A[n/2+1]$ is larger than all n elements in $A[1..n/2] \cup B[1..n/2]$, and therefore larger than the median of $A \cup B$, so we can discard the upper half of A. Similarly, $B[n/2-1]$ is smaller than all $n+1$ elements of $A[n/2..n] \cup B[n/2+1..n]$, and therefore smaller than the median of $A \cup B$, so we can discard the lower half of B. Because we discard the same number of elements from each array, the median of the remaining subarrays is the median of the original $A \cup B$. ■

To think about later:

4. Now suppose you are given two sorted arrays $A[1..m]$ and $B[1..n]$ and an integer k. Describe a fast algorithm to find the kth smallest element in the union $A \cup B$. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1..5] = [2, 5, 7, 17, 19] \qquad k = 6$$

your algorithm should return the integer 7.

Solution: The following algorithm solves this problem in $O(\log \min\{k, m+n-k\}) = O(\log(m+n))$ time:

```
SELECT(A[1..m], B[1..n], k):
  if k < (m+n)/2
      return MEDIAN(A[1..k], B[1..k])
  else
      return MEDIAN(A[k−n..m], B[k−m..n])
```

Here, MEDIAN is the algorithm from problem 3 with one minor tweak. If MEDIAN wants an entry in either A or B that is outside the bounds of the original arrays, it uses the value $-\infty$ if the index is too low, or ∞ if the index is too high, instead of creating a core dump ■

5. Suppose you have an algorithm that given as input a directed graph $G = (V, E)$, nodes $s, t \in V$, and an integer k, outputs whether the *number* of distinct shortest paths from s to t is at least k. Describe an algorithm that counts the number of distinct shortest s-t paths in G. Does your algorithm run in polynomial time?

Solution: This is a Solution sketch. We do binary search again but now we need to upper bound the number of distinct shortest paths from s to t in G. It is not hard to construct examples of graphs where the number is at least $2^{n/2}$ where n is the number of nodes. A crude upper bound is m^n where m is the number of edges and n is the number of nodes. Why? Assuming this upper bound binary search will take $O(m \log n)$ calls and this is polynomial in the input length. Note that writing down the answer may take $O(m \log n)$ bits but that is also polynomial in the input length. ■