Recursion Example - Binary Conversion

- Be careful: computer naturally prints from left to right
  - So we need to first convert N / 2
  - Then write '0' or '1'

```
void bin_convert(int N) {    // 1. Declare function
    if (N == 0) return;      // 2. Base case N=0 -> Stop
    bin_convert(N / 2);      // 3. Convert N/2
    cout << N % 2;           // 4. Print '0' or '1' for
                                   even or odd
}
```

Practice Question: `longestCommonSubsequence`

Write a function named **`longestCommonSubsequence`** that returns the longest common subsequence of both strings.

```
string longestcommonSubsequence(string s1, string s2);
longestCommonSubsequence("mars","megan")  // Returns "ma"
longestCommonSubsequence("chris", "cs32") // Returns "cs"
```

Solution: `longestCommonSubsequence`

```
string longestCommonSubsequence(string s1, string s2) {
  if (s1.length() == 0 || s2.length() == 0) {
    return "";
  } else if (s1[0] == s2[0]) {
    return s1[0] + longestCommonSubsequence(s1.substr(1), s2.substr(1));
  } else {
    string choice1 = longestCommonSubsequence(s1, s2.substr(1));
    string choice2 = longestCommonSubsequence(s1.substr(1), s2);
    return choice1.length() > choice2.length() ? choice1 : choice2;
  }
}
```

Solution: Recursive Merge LL

```
Node* merge(Node* l1, Node* l2) {

   if (l1 == nullptr) return l2;         // base cases: if a list is empty, return the other list
   if (l2 == nullptr) return l1;

   Node* head;                           // determine which head should be the head of the merged list
   if (l1->val < l2->val) {              // then set the next pointer to the head of the recursive calls
      head = l1;
      head->next = merge(l1->next, l2);
   } else {
      head = l2;
      head->next = merge(l1, l2->next);
   }
                                         // return the head of the merged list
   return head;
}
```

Sorting: Quick tips

- Print out Nachenberg's sorting cheat sheet!
- Assuming that we are sorting numbers in increasing order:
  - One pass of Bubble Sort will move the largest item to the end.
  - One pass of Selection Sort will move the smallest item to the start.
  - After n passes of Insertion Sort, the first n items will be in sorted order (as if we completely sorted an array of size n).
  - Bubble Sort, Insertion Sort, and Selection Sort are good for simplicity.
  - Mergesort and Quicksort are good for efficiency.
  - Heapsort and Shellsort are unlikely to be on the exam: understand them and definitely bring notes.

Examples

The following example problems will have this format:

```
[  Original Array  ]
[ After Two Passes ]
```

Q: Which of the following algorithms may have been used?
Assume we are sorting in increasing order.

A. Selection Sort
B. Bubble Sort
C. Insertion Sort
D. More than one of the above

Examples

Q1	**45**	**40**	**42**	**47**	**43**	**41**
	40	41	42	47	43	45

Q2	**45**	**40**	**42**	**47**	**43**	**41**
	40	45	42	47	43	41

Q3	**45**	**40**	**42**	**47**	**43**	**41**
	40	42	43	41	45	47

Solutions

1. A – Selection sort
   a. After 2 passes, first 2 elements of final array are in sorted order
2. C – Insertion sort
   a. After 2 passes, first 2 elements from original array are in sorted order
3. B – Bubble sort
   a. After 2 passes, the last 2 elements of the final array are in sorted order

More Big-O Examples

1. Add n numbers together
2. Find min of n sorted numbers
3. Sort n random integers
4. Find item in STL `set` of m integers
5. Insert an element to beginning of STL `list` of m integers
6. Sort n ages of people

```
Answer Bank

O(1)
O(logn)
O(n)
O(nlogn)
O(logm)
O(m)
O(mlogm)
```

```
Answers:
O(n), O(1), O(nlogn), O(logm), O(1), O(n)
```

Solution: Find the maximum depth of a BST

```
int maxDepth(Node* root) {
  if (root == nullptr) { return 0; }       // If no node, then depth is zero.
  int leftSum = 1 + maxDepth(root->left);  // Add 1 to account for the root node.
  int rightSum = 1 + maxDepth(root->right);
  return (leftSum >= rightSum) ? leftSum : rightSum;
}
```

Solution: Return if a tree is balanced

```
#include<cmath>
bool isBalanced(Node* root) {
  if(root == nullptr) { return true;}
  int L = maxDepth(root->left);
  int R = maxDepth(root->right);
  int diff = abs(L - R);
  return diff <= 1 &&
      isBalanced(root->left) &&
      isBalanced(root->right);
}
```

```
int maxDepth(Node* root) {
  if(root == nullptr) { return 0; }
  int leftSum =
      1 + maxDepth(root->left);
  int rightSum =
      1 + maxDepth(root->right);
  return (leftSum >= rightSum) ?
      leftSum :
      rightSum;
}
```

Writing hash functions

```
a[n] * pow(31, n) + a[n - 1] * pow(31, n - 1) + ... + a[0]
```

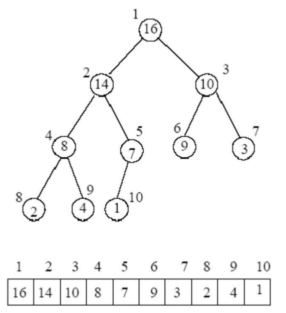
- Previously, we gave a slightly simplified model for hash functions!
- On top of providing a corresponding integer for a given key, we need to apply the modulo operator to find an actual bucket.
- When writing a hash function for strings, consider using the position as well as the character value itself when computing a corresponding integer.

Comparison with binary search trees

	Binary Search Tree ***Ordered!***	**Hash Table** ***Unordered!***
Access	`O(log(n))`	`O(1)`
Search	`O(log(n))`	`O(1)`
Insertion	`O(log(n))`	`O(1)`
Deletion	`O(log(n))`	`O(1)`

Heaps

- A heap is just a modified tree data structure that satisfies either the max or min heap property
- Max heap: the parent node's value is greater than that of its children (shown on the right)
- Min heap: the parent node's values is less than that of its children
- A heap can be visually represented as a tree!



Lets consider the *i*th **node** in a Heap that has the value A[i]

`PARENT(i) = i/2`	Return the index of the father node
`LEFT(i) = 2i`	Return the index of the left child
`RIGHT(i) = 2i+1`	Return the index of the right child

Priority Queue

- A priority queue is an abstract data type like a queue or stack, but with priority associated with its element
- An element with high priority is served before an element with low priority
- A priority queue can be implemented with heaps
- Provides constant lookup of highest-priority object, with expense of logarithmic insertion/extraction
- For STL priority_queue, use `top()`, `pop()`, and `push(v)`

```
int values = [1,8,5,6,3,4,0,9,7,2];

priority_queue<int> q;
for(int i = 0; i < 10; ++i)
    q.push(values[i]);
print_queue(q); // A function we write
// 9 8 7 6 5 4 3 2 1 0

priority_queue<int, vector<int>,
greater<int> > q2;
for(int i = 0; i < 10; ++i)
    q2.push(values[i]);
print_queue(q2);
// 0 1 2 3 4 5 6 7 8 9
```

Set, map, priority queue caveats.

For user-defined classes, the `set`, `map`, and `priority_queue` classes all rely on some form of custom comparator to know how to order its elements:

```
struct Comparator {
  // Returns `true` if lhs should be ordered before rhs.
  bool operator() (const Student& lhs, const Student& rhs) const {
    return lhs.get_id() < rhs.get_id();
  }
};
```

Unordered set, map caveats.

- For user-defined classes, the `unordered_set` and `unordered_map` classes rely on some form of hash function to determine its buckets.
- They also need to be able to determine equality between elements.

C++ Minheap

Suppose we want to be able to declare a minheap more easily:

```
minheap<int> heap;
```

How would you define the minheap type?

C++ Minheap

Using a priority queue:

```
template<typename T>
using minheap = priority_queue<T, vector<T>, greater<T>>;
```

`using` means that we are declaring this type in terms of another type.
`vector<T>` indicates the underlying data structure.
`greater<T>` indicates that **later** values to be popped from the heap will be greater.

Search strategies: depth-first search

```
Imagine we have an itinerary.
Add our start location to the end of the itinerary.
While our itinerary still has places to visit:
    Grab the last unvisited place we added to the itinerary.
    Move here, and mark this place as visited.
    If this is our goal, we're done.
    If not, add all places around us to the end of our itinerary.
```

What data structure might be good for this?

Search strategies: depth-first search

- Depth-first search is *very space efficient!* If you implement it recursively, it only needs to store a single path in memory.
- However, depth-first search is not guaranteed to find the optimal solution.
- Furthermore, depth-first search can potentially fail to terminate in the case that it encounters an infinitely long path.

Search strategies: breadth-first search

- We can implement breadth-first search using a queue.
- Unlike the depth-first search, the breadth-first search is guaranteed to find the shortest path.
- Unfortunately, it keeps track of all existing paths, which means that its memory footprint can grow very quickly.

Pseudocode: Depth First Search

```
stack = Stack()
stack.push(newPath(startNode))
seen = Set()
while !stack.isEmpty()
  currPath = stack.pop()
  currState = last(currPath)
  if(currState is goal)
    return currPath
  if(seen contains currState)
    continue
  seen.add(currState)
  for nextState in getNextStates(currState)
    path = newPath(currPath, nextState)
    stack.push(path)
```

Pseudocode: Breadth First Search

```
queue = Queue()
queue.enqueue(newPath(startNode))
seen = Set()
while !queue.isEmpty()
  currPath = queue.dequeue()
  currState = last(currPath)
  if(currState is goal)
    return currPath
  if(seen contains currState)
    continue
  seen.add(currState)
  for nextState in getNextStates(currState)
    path = newPath(currPath, nextState)
    queue.push(path)
```