|  |  |  |
| --- | --- | --- |
| **Insertion Sort**  void insertion\_sort(int arr[], int length)  {  int i, j ,tmp;  for (i = 1; i < length; i++) {  j = i;  while (j > 0 && arr[j - 1] > arr[j]) {  tmp = arr[j];  arr[j] = arr[j - 1];  arr[j - 1] = tmp;  j--;  }  Worst Case: O(n^2) Best Case: O(n) | **Bubble** **Sort**  void bubble\_sort(int arr[], int size) {  bool not\_sorted = true;  int j=1,tmp;    while (not\_sorted) {  not\_sorted = false;  j++;  for (int i = 0; i < size - j; i++) {  if (arr[i] > arr[i + 1]) {  tmp = arr[i];  arr[i] = arr[i + 1];  arr[i + 1] = tmp;  not\_sorted = true;    }//end of if  print\_array(arr,5);  }//end of for loop  }//end of while loop  }//end of bubble\_sort  Worst Case: n-1 passes O(n^2) Best Case: one pass, no swaps O(n) | **Selection Sort**  for (int i = 0; i < (N - 1); i++)  {  int minIndex = i;  // Find the index of the minimum element  for (int j = i + 1; j < N; j++)  {  if (a[j] < a[minIndex])  {  minIndex = j;  }  }  // Swap if i-th element not already smallest  if (minIndex > i)  {  swap(a[i], a[minIndex]);  }  }  Worst Case: O(n^2) Best Case: O(n^2) |
| **Merge Sort**  Worst Case: O(nlogn) Best Case: O(nlogn) | **Quick Sort**  Worst Case: O(n^2) Best Case: O(nlogn) | **Heap Sort**  Worst Case: O(nlogn) Best Case: O(nlogn) |

|  |  |
| --- | --- |
| **Advantages of Templates**: generality. Write one list, get all data types free.  Completely general? No. We need to be able to compare the data we’re storing:  while (cursor != nullptr && cursor->next != nullptr && cursor->next->data < payload) | **Disadvantages of Template**:  \* Can’t compile without a main program  \* Give lots more distressing error messages for same number of errors  \* Not always implemented efficiently under the hood |

1. Show me how the type of a **template class gets defined** in the main() program.

|  |  |
| --- | --- |
| struct node {  ItemType data; // generic data type  node\* next;  node\* prev;  }; | Int main()  {  node<int>\* head\_ptr1 = nullptr; } |

1. How would you define an array of big\_numbers using a **template array class** like the ones from the sorting codes?
   1. In main: node<big\_number>\* head\_ptr1=nullptr; ???
2. How could you count the frequency of a number in a **binary search tree**?
   1. If you traverse the tree [InOrder](http://en.wikipedia.org/wiki/Tree_traversal" \l "Inorder_Traversal) you'll get the repeated elements together, so you just need to check when a value equals the previous, and how many times this happens.
3. How could you tell if 2 **binary search trees** contained exactly the same numbers?
   1. Use either preorder, inorder, or postorder traversal and compare each number. Also, the shape of both trees will be the same.
4. Could we make a **template class for binary search trees**? Why or why not?
   1. Yes to a BST of strings, numbers, or big\_nums, but you can’t use comparison operators on an item type of BSTs.
5. What’s the expected run time for **quicksort**? Why? What’s the worst run time it can get, and when does this happen?
   1. Quicksort is usually O(n log n), but it can do worse. It depends on the pivot choice, it must be a value in the middle range of the array, gotten by taking the median of the first, middle, and last element. We make this choice carefully because if we’re unlucky in every pivot selection, the algorithm degenerates to O(n^2) again.
6. What’s the expected run time for **mergesort**? Why? What does mergesortdo that’s inefficient compared to the other sorting algorithms?
   1. O(nlog2n). The downside of mergesort is that each of the little merges requires us to make an extra array to hold the merged data. We copy this array back at each step to the main array and write the merges into the second array. The overhead of this extra array is pretty expensive, for large n.
7. What’s the best case run time for **insertion sort**? What produces this run time? What’s the expected run time for insertion sort? What about **bubble sort** and **selection sort**?
   1. Best case for insertion sort and bubble sort is O(n). This only occurs if the array is already sorted and no swaps need to be made. Expected run time is O(n^2).
   2. All cases of selection sort are O(n^2)
8. Convert 23415 in base 7 to base 5, using the digit-wise algorithm from class.
9. What is the problem with this code?

int& no\_no\_nanette() {

int answer = 9;

return answer;

}

Int answer is a **dangling reference**. These are bad because as soon as the function returns, answer goes out of scope; it does not exist outside of the function. Now the calling function has a reference variable that points to nowhere.

Why is that not a problem with this code?

big-number& operator =(const big\_number& m) {

…

return \*this;

}

big\_number m is a constant variable, and it exists globally (outside the function). When the function ends, the variable does not go out of scope.

1. Tell me 2 big differences between a **copy constructor** and an **assignment operator** (operator =). Justify your answers.
   1. A copy constructor is used to initialize a previously uninitialized object from some other object's data. An assignment operator is used to replace the data of a previously initialized object with some other object's data.
   2. Don’t write a copy constructor if shallow copies are ok: In C++, If an object has no pointers or any run time allocation of resource like file handle, a network connection..etc, a shallow copy is probably sufficient. Therefore the default copy constructor, default assignment operator, and default destructor are ok and you don’t need to write your own.
2. Given the code for **operator >**, what could you do to get **operator <** for very little work? How would you get **operator ==** from the > and < operators?
   1. !(a<b) && a==b
   2. !(a < b) && !(b < a) or !(a<b) && !(b<a)
3. For the pattern code, how many **pattern calls** result from calling:

pattern(outs, 4, 0); 2?

pattern(outs, 16, 0); 4?

pattern(outs, 1024, 0); 10?

What formula describes this relationship between the starting n and the number of calls **pattern** makes?

1. What formula describes the relationship between the starting n and the number of stars **pattern** prints?
   1. N must be a positive power of two.
2. Show me the code that checks for self assignment in operator =, and tell me what it’s checking.

Where else in big\_number might we need to check for this?

1. What does partition do to make the array ‘less unsorted’ than before?
   1. It splits the array into two smaller arrays at the pivot point. The first of which contains all values <= to the pivot, the second contains all values > the pivot.
2. Given an array representation of a heap, tell me a formula to get the parent of a heap item in the array. Tell me a formula to get an array heap item’s left child, and its right child.
   1. Left child: 2n+1
   2. Right child: 2n+2
   3. Parent: (n-1)/2
3. Given the answers to the question above, write me a loop that re-heapifies a heap after one new element has been added.
4. Given the answers to the question above, write me a loop that re-heapifies a heap after the root element has been removed.