Basic Algorithm Analysis

- There are often many ways to solve the same problem.
- We want to choose an efficient algorithm.
- How do we measure efficiency? Time? Space?
- We don't want to redo our analysis every time we reimplement our algorithm, or use a different machine.

Basic Algorithm Analysis

- To analyze the amount of time required, we count the number of operations needed.
- But not all operations require the same amount of time.
- Different compiler/machine may translate the same C++ code to different number/type of instructions.
- Instead, we pick one "representative operation" and count that (e.g. comparisons, variable assignments).
- The idea is that the total amount of time is **proportional** to the number of representative operations. The constant of proportionality depends on compiler, machine, etc.
- That is, the analysis is dependent **only** on the algorithm, not on the implementation.

Big-O Notation

- We express the number of operations in “big-O” notation.
- The number of operations depends on the size of input (e.g. number of array elements). Usually denoted n.
- Roughly, “big-O” means “proportional to” for **large enough** inputs.
- e.g. $O(n)$ means run time is proportional to input size. Doubling the input means roughly doubling the running time.
- e.g. $O(n^2)$ means run time is proportional to the square of input size. Doubling the input means roughly quadrupling the running time.
- We want an algorithm with a small “big-O”.
- “Run time” is usually called the **(time) complexity**.

Worst case vs. Average Case

- Some algorithms behave differently depending on input.
- For example, some sorting algorithms will be very fast if the input is already sorted, but they may be much slower on other cases.
- We can talk about the worst case complexity: how fast will the algorithm run regardless of what input it receives?
- We can also talk about average case complexity: how does it do on average?
- Average case is hard: we need to know what "average" means.
- We will concentrate on worst case complexity.

Searching: Linear Search

```
int find(int A[], int n, int key)
{
  for (int i = 0; i < n; i++)
    if (A[i] == key)
      return i;
  return -1;  // not found
}
```

- In the worst case (key is in the last position/not found), we need to do n comparisons.
- The algorithm has $O(n)$ complexity.

Searching: Binary Search

```
int find(int A[], int n, int key)
{
  int lo = 0, hi = n-1;
  while (lo <= hi) {
    int mid = (lo + hi) / 2;
    if (A[mid] == key)
      return mid;
    else if (key < A[mid])
      hi = mid-1;
    else
      lo = mid+1;
  }
  return -1;  // not found
}
```

Searching: Binary Search

- The array must be sorted.
- The worst case happens if the key is not found.
- Every time the size of the range `[lo,hi]` is reduced by at least half.
- That means it takes approximate $\log_2 n$ iterations to reduce `[0,n-1]` to the empty range.
- Each iteration does a "constant" amount of work, so the algorithm has $O(\log_2 n)$ complexity.
- We usually write $O(\log n)$ because logarithms of different bases are related by a constant: $\log_2 n = \log n / \log 2$

Complexity: Does It Matter?

- For large n, the difference between $O(n)$ and $O(\log_2 n)$ is big. The number of iterations:

n	$O(n)$	$O(\log_2 n)$
32	32	5
1024	1024	10
1048576	1048576	20

Sorting: Selection Sort

```
for (int i = 0; i < n; i++) {
  int index = find_min(A, i, n);
  swap(A[i], A[index]);
}
```

- Finding the minimum on n elements requires $O(n)$ comparisons.
- With n iterations this becomes $O(n^2)$.
- More precisely, the i-th iteration requires about $n-i$ operations, so the total is:

$$n + (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n+1)}{2} = O(n^2)$$

- $O(n^2)$ is slow. Sorting 10^6 entries requires roughly 10^{12} operations!

Other "Slow" Sorting Algorithms

- The following common sorting algorithms are also $O(n^2)$:
  - Bubble sort
  - Insertion sort
- But these algorithms can be $O(n)$ if the array is already sorted.
- Selection sort is **always** $O(n^2)$ even if the array is already sorted.

Merge Sort

- This is our first fast sorting algorithm.
- The basic idea is this:
  - If the array has only one element, it is easy.
  - Otherwise, split the array into two halves.
  - Recursively sort each half.
  - Merge the two sorted lists together.
- The only “real work” is in the merging.

Merge Sort

```
void mergesort(int A[], int start, int end)
{
  if (end - start > 1) {
    int mid = (start+end)/2;
    mergesort(A, start, mid);
    mergesort(A, mid, end);
    merge(A, start, mid, end);
  }
}
```

Merging

Merging is done with a "marching algorithm":

```
void merge(int A[], int start, int mid, int end)
{
  int R[SIZE], i1, i2, j;
  i1 = start;  i2 = mid;  j = 0;
  while (i1 < mid && i2 < end) {
    if (A[i1] < A[i2])  // A[i1] comes next
      R[j++] = A[i1++];
    else
      R[j++] = A[i2++];
  }
  copy(A+i1, A+mid, R+j);
  copy(A+i2, A+end, R+j+(mid-i1));
  copy(R, R+(end-start), A+start);
}
```

Merging

- Notice that each loop iteration copies one element from either half.
- The two `copy()`'s merge the leftovers.
- Each element is copied once into `R` and once back into `A`.
- So merging has complexity $O(\text{end} - \text{start})$.
- Notice that we need an auxillary array.

Merge Sort: Complexity

- At the top level, merging takes $O(n)$ operations.
- At the next level, merging takes $O(n/2)$ operations on each half, but there are two halves. Total: $O(n)$.
- The next level works on quarters: $O(n/4)$ for merging, but there are 4 quarters. Total: $O(n)$.
- Every level requires $O(n)$ work.
- How many levels are there?
- Each level the size is reduced by half: $O(\log_2 n)$ levels
- Total complexity: $O(n \log n)$.

Quicksort

- A very common “fast” sorting algorithm.
- It is commonly considered to be one of the fastest algorithms.
- Same complexity as merge sort on average, but “proportionality constant” is much smaller.
- Requires relatively little extra space.
- Like mergesort, quicksort is recursive.

Quicksort

The idea:

- If there is one element or less, the array is sorted.
- Otherwise, choose a **pivot** element.
- **Partition** the array so all elements to the left of the pivot are no bigger than the pivot, and all elements to the right are larger.
- Recurse on the subarrays to the left and right of the pivot.

The only real work is done in the partitioning.

Quicksort

```
void quicksort(int A[], int start, int end)
{
  if (end - start > 1) {
    int pivot = partition(A, start, end);
    quicksort(A, start, pivot);
    quicksort(A, pivot+1, end);
  }
}
```

Partitioning

- The pivot can be any element in the array. It is easiest to choose the first one (`A[start]`)
- One way: scan from left to right, and maintain two indices: `i` and `j` so that
  - `A[start+1..i)` are $\leq$ `A[start]`
  - `A[i,j)` are $>$ `A[start]`
- Initially, `i = j = start+1`.
- We go through `j = start+1` to `end`.
- If `A[j] > A[start]`, just increment `j`.
- Otherwise, swap `A[j]` and `A[i]` and increment both `i` and `j`.
- At the end, swap `A[start]` and `A[i-1]`.

Partitioning

```
int partition(int A[], int start, int end)
{
  int i, j;
  i = start+1;
  for (j = start+1; j < end; j++)
    if (A[j] <= A[start])
      swap(A[i++], A[j]);
  swap(A[start], A[i-1]);
  return i-1;   // return where the pivot is
}
```

Complexity

- The sizes of the two subarrays depend on the choice of pivot.
- If we are lucky, the two subarrays are roughly half the size of the original.
- Since partition takes $O(n)$ operations, quicksort has $O(n \log n)$ complexity (same reasoning as merge sort).
- If we are unlucky, the pivot is the smallest/largest element. Since we reduce the size by 1, we need $O(n)$ levels and the complexity is $O(n^2)$.
- The worst case of quicksort happens when the array is already sorted (or sorted in reverse)!
- On average, the complexity is $O(n \log n)$.

Another Partitioning Algorithm

We can move from both ends:

```
int partition(int A[], int start, int end)
{
  int pivot = A[start];
  int i = start+1, j = end-1;
  while (i <= j) {
    while (A[i] <= pivot) i++;
    while (pivot < A[j]) j--;
    if (i < j) swap(A[i], A[j]);
  }
  swap(A[start], A[j]);
  return j;
}
```

This is slightly more efficient but more difficult to get correctly (still $O(n)$).

Another Partitioning Algorithm

Did you see the bug(s)?

Optimizations

- Instead of choosing the first element as pivot, we can choose a random element. It is unlikely to be the smallest/largest.
- Another common approach: pick three elements and use the middle one as pivot.
- "Fat pivot": partition array into three subarrays: less than the pivot, equal to the pivot, and greater than the pivot. If there are many equal elements, the left and right subarrays are smaller.
- Recursion has overhead (function calls): it is worthwhile only for large arrays. When the size of subarray is small enough, use a different sorting algorithm (e.g. insertion sort).
- The "cutoff" point to switch between algorithms depends on compiler, machine, etc. It needs to be tuned experimentally.

Generic Sorting

- The STL sorting algorithms are "generic": it works as long as `operator<` is defined on the elements (or with a supplied comparison function).
- Also, it uses the standard STL iterators to specify the range (random access iterators).
- Our code can be easily adapted for templates:
  - Instead of comparing by `<=`, use OR of `<` and `==` (or use `less_equal` from `<functional>`).
  - Since the start and end iterators are random access, we can use iterator arithmetic to jump to particular elements in the range.

Data Structure: Heaps

- A **heap** is a binary tree with the following properties:
  - each element in the heap is $\geq$ its two children (if they exist);
  - the tree is completely filled on all levels except possibly the lowest, which is filled from the left.
- Because of this property, we can represent a heap of n elements in an array of size n, such that:
  - The two children of node i is in positions $2i + 1$ and $2i + 2$.
  - The parent of node i is in position $\lfloor (i - 1)/2 \rfloor$.
- **Note**: the word “heap” is also used to describe dynamically allocated memory. This is a completely different use of this word.

Heaps: Sifting Up

Suppose we have a heap `A` of size $n-1$ and we want to add an extra element to `A[n-1]`. We can fix up the heap by "sifting up":

```
void sift_up(int A[], int n)
{
  int i;
  for (i = n-1; i > 0 && A[i] > A[(i-1)/2]; i = (i-1)/2)
    swap(A[i], A[(i-1)/2]);
}
```

In other words, if `A[i]` is larger than its parent, we swap `A[i]` with its parent and repeat.

Heaps: Sifting Up

The sifting up procedure is correct, because:

- We assume that the array has the heap property except possibly with `A[i]` and its parent (true at the beginning).
- At each iteration, if the heap property is violated, we swap `A[i]` with its parent. Otherwise, we are done.
- If `A[i] > A[(i-1)/2]`, then after swapping `A[(i-1)/2]` is still $\geq$ its children.
- We can also show that after swapping `A[i]` is still $\geq$ its children (we won't prove this formally).

Heaps: Sifting Up

- The complexity is the height of the tree, which is $O(\log n)$.
- To build a heap, we start from a heap of size 1 and repeatedly sift up additional elements:

```
for (i = 2; i <= n; i++)
  sift_up(A, i);
```

- This has complexity $O(n \log n)$.
- More precisely, it is $O(\log 2 + \log 3 + \cdots + \log n)$, which simplifies to $O(n \log n)$, but some math is needed to understand this...

Heaps: Sifting Down

- Suppose that we change the first element `A[0]` in a heap.
- Then the heap property is preserved except perhaps between `A[0]` and its children (i.e. `A[0]` is too small).
- We can fix it by "sifting down": swap it with the larger of the two children.
- Because we choose the larger child, the heap property is restored except for the heap rooted at the swapped child.
- Move down and repeat.
- Again, the complexity of each sifting down operation is $O(\log n)$.

Heaps: Sifting Down

```
void sift_down(int A[], int n)
{
  int i = 0;
  while (2*i+1 < n) {   // there is a child
    int child = 2*i+1;  // left child
    if (child+1 < n && A[child] < A[child+1])
      child++;          // check right child, if it exists
    if (A[i] >= A[child])
      break;            // heap property is good
    swap(A[i], A[child]);
    i = child;
  }
}
```

Heaps: Extracting the Maximum

- The maximum is at the root of the heap.
- To remove the maximum, just put `A[n-1]` into the root and sift down (after decrecmenting `n`).
- Complexity is $O(\log n)$.

Priority Queues

- Heaps can be used to implement priority queues.
- Insertion has $O(\log n)$ complexity.
- Deleting the maximum has $O(\log n)$ complexity.
- Finding the maximum element (without deleting it) has $O(1)$ complexity (i.e. constant time).

Heap Sort

A heap can be used to sort an array of n elements as follows.

- Build a heap with the n elements: $O(n \log n)$.
- Repeatedly extract the maximum of the heap and put the maximum at the correct spot. Decrease the size of the heap by 1. $O(\log n)$ each iterations gives $O(n \log n)$.
- At any time, the first portion of the array is a heap, the second portion is the sorted list.
- Complexity is $O(n \log n)$.

Heap Sort

```
void heapsort(int A[], int n)
{
  // build the heap
  for (int i = 2; i <= n; i++)
    sift_up(A, i);

  for (int i = n-1; i >= 1; i--) {
    swap(A[0], A[i]);
    sift_down(A, i);
  }
}
```

Sorting: Can we do better?

- The fastest sorting algorithm we have seen has $O(n \log n)$ complexity.
- If the data to be sorted is "special", we can do better.
- e.g. if we are sorting 0 and 1's, we can just count the number of 0's and number of 1's. This has $O(n)$ complexity.
- "Sorting by counting" is fast as long as the number of different values is "small".
- We cannot do better than $O(n)$ because we have to look at every array element.

Sorting: Can we do better?

- It can be proved that if you sort by comparing elements, it is **impossible** to have an algorithm that has a better worst case complexity than $O(n \log n)$ (take CS 3620 if you want a precise statement).
- It does not matter how smart you are. "Impossible" means impossible.
- It means that if you have any sorting "algorithm" which does better than $O(n \log n)$ in the worst case, either:
  - the complexity analysis is wrong, or
  - there is at least one input array on which your "algorithm" gives the wrong answer.