Time Complexity

Gauging Large-N Performance
Complexity

- Many programs are written for and designed to be used on a large scale.
  - Banks can have millions of customers.
  - Search engines index billions, even trillions, of pages.
  - Map + GPS software find routes for both small and large trips.
    - There are likely millions of roads in the US.
Complexity

• On such large scales, the performance of an algorithm begins to matter greatly.
  – In particular, the growth rate of the execution time (or space, in some scenarios) is critical to examine.
• For this lecture, at least, we will only consider time complexity.
Complexity Notation

• First, some notation: “Big-Oh”.
  – $T(N)$ is $O(f(N))$ if, for some positive constants $c$ and $N_0$, $T(N) \leq c f(N)$ whenever $N \geq N_0$.
  – That is, the running time (for $N$ elements) of an algorithm is “order $f(N)$” if, for some positive constants....
Complexity Notation

• Even if the constants in question are large for \( f(N) \), as long as this condition is met, the corresponding algorithm is considered to be \( O(f(N)) \).
  – The core idea of “big oh” notation is to establish an upper bound on an algorithm’s running time.
Complexity Notation

• It is “$O(f(N))$” which is most often referred to when talking about a method being “order $f(N)$.”
  – Perhaps this is because for many, more complex algorithms there is no known lower bound.
  • Not all problems are known to be optimally solved in regard to time complexity!
Complexity Notation

• All of these exist to describe an algorithm’s *asymptotic* behavior – the overall behavior of the growth rate.

• Like the asymptote in a mathematical expression, only the highest-order term matters.
Complexity Notation

• If an algorithm is “$O(N^2 + N)$”, it is automatically $O(N^2)$.
  – It would be *incorrect* to say the former. The method *is* $O(N^2)$.

• Likewise, if an algorithm is “$O(2N^2)$”, it is automatically $O(N^2)$.
  – It’s incorrect to place known constants in the “order of” expression.
Complexity Notation

• However, it is completely allowable to include more than one variable in the “order of” expression.
  – This can matter for algorithms on complex data structures.
Complexity Notation

• A reminder about “Big-Oh” notation.
  – $T(N)$ is $O(f(N))$ if, for some positive constants $c$ and $N_0$, $T(N) \leq c f(N)$ whenever $N \geq N_0$.
  – In such situations, we often say that the running time (for $N$ elements) of an algorithm is “order $f(N)$.”
Algorithm Evaluation

• Now that we’ve seen some of the basic notation of time complexity, let’s move on to how we can use it to evaluate the performance of algorithms and associated data structures.
int sum(int* a, int cnt)
{
    int s = 0;

    for(int i=0; i < cnt; i++)
    {
        s += a[i];
    }

    return s;
}
A Basic Example

```c
int sum(int* a, int cnt)
{
    int s = 0;
    for(int i=0; i < cnt; i++)
    {
        s += a[i];
    }
    return s;
}
```

Let $N = cnt$, input as the length of the array $a$. Given this, how many operations will be needed to evaluate this algorithm?
A Basic Example

```c
int sum(int[] a, int cnt)
{
    int s = 0;
    for (int i = 0; i < cnt; i++)
    {
        s += a[i];
    }
    return s;
}
```

Operations needed to get the ith element of the array a: 1. (Or, at least, O(1).)

Ops needed for a single “+=”: 1.
A Basic Example

```c
int sum(int a[], int cnt)
{
    int s = 0;
    for(int i=0; i < cnt; i++)
    {
        s += a[i];
    }

    return s;
}
```

Each increment step ("i++"): 1 op
Each "test" step of the for-loop: 1 op
A Basic Example

```c
int sum(int[] a, int cnt)
{
    int s = 0;
    for (int i = 0; i < cnt; i++)
    {
        s += a[i];
    }
    return s;
}
```

So, each iteration of this for-loop takes 4 operations.

4 ops/item * N items = 4N ops for the loop.
A Basic Example

• Since this algorithm takes $4*N$ operations for the entire loop over $N$ items, and all the other code lines take constant time, this is an $O(N)$ algorithm.
Binary Search

- The *binary search* technique is designed to find items within a pre-sorted list.
  - The technique is recursive in nature, though it can easily be written in iterative form.
template <typename T>
int binarySearch (T* array, T item, int begin, int end)
{
    int mid = (end + begin) / 2;
    if(array[mid].equals(item))
        return mid;
    else if(begin == end)
        return -1;
    else if(array[mid] > item)
        return binarySearch(array, item, mid+1, end);
    else //if(array[mid] < item)
        return binarySearch(array, item, begin, mid-1);
}
Binary Search

• For an array of size 1, the needed time is constant – $O(1)$.
  – Let’s call this constant C.

• For an array of twice the size, one more (recursive) call of the method will be evaluated.
  – That’s an extra C added to the computation time.
Thus, for an array of size $N$, let us note the following:

\[
T(N) = C + T\left(\frac{N}{2}\right)
\]
\[
= 2C + T\left(\frac{N}{4}\right)
\]
\[
= \ldots
\]
\[
= C \times (\log_2(N) - 1) + T(1)
\]
\[
= C \times \log_2(N)
\]

Binary search is thus $O(\log(N))$. 
Data Structures

• Thus far, we’ve looked at time complexity in regard to a few algorithms on basic arrays.
  – Arrays have $O(1)$ time complexity for storing and retrieving items.
  – However, not every data structure has the same complexity for these operations.
Data Structures - Lists

- To remove an item from a random location within a vector will also take \( O(N) \) time.
  - This is because the backing array must move \( O(N) \) items so that a giant hole isn’t left in the middle of the list.
• However, this isn’t the only structure we’ve seen that can model a list... we should also examine the list structure. (C++’s linked list)
  – `operator[]` will take $O(N)$ time for a randomly positioned item.
Data Structures - Lists

- `push_back()` will take $O(1)$ time, as would a `push_front()`, or a `pop_back()`.
  - However, $O(N)$ time would be required for a random position insert.
  - This is because it takes $O(N)$ time to iterate to the position at which the element will be inserted.

- Random-placement removal will generally take $O(N)$. 
Data Structures - Lists

- The methods designed to operate on the first and last elements will take constant time – those can be accessed directly.
Data Structures - Lists

• However, to *iterate* over an entire vector with its iterator will take only $O(N)$ time.
  – That’s only $O(1)$ per item.
  – The iterator has access to the internals of the data structure, allowing it to track its position within the list.
Data Structures - Lists

• Note that the iterator class provides a removal-type method.
  – While to call remove() on a random element within a list will take $O(N)$ time, calling remove() on an element of a list from its iterator will take only $O(1)$ time.
## Data Structures - Lists

<table>
<thead>
<tr>
<th>Operation</th>
<th>Vector</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>get, set (random)</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>add (random)</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>remove (random)</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>add (last)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>add (first)</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove (last)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove (first)</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
## Data Structures - Lists

<table>
<thead>
<tr>
<th>Operation</th>
<th>vector</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>iteration (whole list)</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>iteration (per item)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove (while iterating)</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Other Algorithms

• The binary search algorithm is hardly the only method whose time complexity analysis involves a recurrence relation.
  – Merge sort, quick sort...
  – map operations...
Merge Sort

- For the operation of merge sort, we know that the array being sorted is divided in half on each recursive frame.
  \[ T(N) = 2 \times T \left( \frac{N}{2} \right) + ??? \]

- The remaining question: how much time is required per frame for its non-recursive operations?
Merge Sort

- Merge sort must (in some manner) divide its array in half.
  - If the array is copied into two separate arrays, the division time is $O(N)$.
  - If the array is left in-place, with the division demarked by ints, the division time is $O(1)$. 
Merge Sort

• After the two recursive halves return, merge sort must then merge the two halves together.
  – The merge operation requires $O(N)$ time to complete, per recursive frame of the entire method’s operation.
Merge Sort

• The recurrence relation for merge sort is then of the form

\[ T(N) = 2 \times T\left(\frac{N}{2}\right) + O(N). \]

• Now, the remaining problem is how get the closed-form for this recurrence relation.
Merge Sort

\[-T(N) = 2 \times T\left(\frac{N}{2}\right) + O(N)\]
\[= 4 \times T\left(\frac{N}{4}\right) + O(N) + O(N)\]
\[= 8 \times T\left(\frac{N}{8}\right) + O(N) + 2O(N)\]
\[= 16 \times T\left(\frac{N}{16}\right) + O(N) + 3O(N)\]
\[= K \times T\left(\frac{N}{K}\right) + \log_2(K) \times O(N),\]
where \(K = 2^p, \ p \in \mathbb{Z}^+\).
Merge Sort

\[-T(N) = 2 \times T\left(\frac{N}{2}\right) + O(N)\]
\[= 4 \times T\left(\frac{N}{4}\right) + O(N) + O(N)\]

Note – there’s some definite abuse of notation going on here with the \(O(N)\)s – it’s complicated by the shrinkage of \(N\) as the recursion gets deeper.

where \(K = 2^p, \ p \in \mathbb{Z}^+\).
Merge Sort

\[ T(N) = K \cdot T \left( \frac{N}{K} \right) + (2 \log_2(K) - 1) O(N) \]

where \( K = 2^p, \ p \in \mathbb{Z}^+ \ldots \)

- If \( K = N \) – if \( N \) were a power of two, this gives \( T(N) = N + \log_2 N \cdot O(N) \), which reduces down to
  \[ T(N) = O(N \log_2 N). \]