Today …

• Homework 5
• More on Algorithm Analysis
• Continuing with Sorting
• Mergesort

[Sec. 9.2, pp. 466-472]
Algorithm Analysis:
Constant Factors

Big-O Notation
We drop the constant factors and dominated terms

– Instead of $O(6n - 5)$ we just write $O(n)$

– Instead of $n^2 - 5n + 10$ we just write $O(n^2)$

Why can we do this?
Big-O Notation

Recall from the definition of Big O that if

- Algorithm A’s worst-case running time is \( f(n) \) and order \( g(n) \)
- there are constants \( k \) and \( n_0 \) s.t. \( f(n) \leq k \cdot g(n) \) for \( n \geq n_0 \)

Note that \( n \) is always greater than or equal to 0 … \( n \geq 0 \)

• For example, suppose A’s “nitty gritty” worst-case running time is 27 … i.e., \( f(n) = 27 \)
  - we say that \( A \) is \( O(1) \)

  If we pick \( n_0 = 0 \) (any value will work) we have that
  - \( 27 \leq k \cdot 1 \) for \( k = 27 \) (actually, for \( k \geq 27 \))

Big-O Notation

• Suppose A’s detailed worst-case running time is:

\[ 6n + 3 \quad \ldots \quad \text{i.e., } f(n) = 6n + 3 \]

  - we say that \( A \) is \( O(n) \)

• What values do we pick for \( n_0 \) and \( k \) to verify this?

  Lets try \( n_0 = 1 \) which gives

  \[ f(1) = 6 \cdot 1 + 3 \leq k \cdot 1 \]

  - In this case, setting \( k = 9 \) does the trick!
  - Note that for \( n \geq 1 \), \( 6n + 3 \leq 9n \) always holds!

  We could use other values for \( n_0 \) as well here …
Big-O Notation

Here are the graphs of these functions

![Graph of functions](image)

### Big-O Notation

- Suppose A’s detailed worst-case running time is
  \[ f(n) = n^2 - 5n + 10 \]
  - we say that A is \( O(n^2) \)

- What values do we pick for \( n_0 \) and \( k \) to solve this?

  We want to show that
  \[ n^2 - 5n + 10 \leq k \cdot n^2 \text{ for some } n_0 \]

  Let’s try \( n_0 = 2 \) which gives
  \[ f(2) = 4 - 2 \cdot 5 + 10 \leq k \cdot 4 \]
  - in this case, setting \( k = 1 \) does the trick!
Types of Sorting Algorithms

- **Exchange-based**
  - Swap pairs of items
  - Bubble sort, cocktail sort, comb sort (1980)

- **Selection-based**
  - Select smallest item and move into place
  - Selection sort, heapsort (1964)

- **Insertion-based**
  - Insert new items into already sorted lists
  - Linear insertion, shell (1959), tree (1964), library sort (2004!)

- **Partitioning- and Merging-based**
  - Divide and conquer (break problem into 2 or more subproblems)
  - Mergesort (1945), quicksort (1962)
Types of Sorting Algorithms

- **Exchange-based**
  - *Swap pairs of items*
  - *Bubble sort*, *cocktail sort*, *comb sort* (1980)

- **Selection-based**
  - *Select smallest item*
  - *Selection sort*, *heapsort* (1964)

- **Insertion-based**
  - *Insert new items into already sorted lists*

- **Partitioning- and Merging-based**
  - *Divide and conquer (break problem into 2 or more subproblems)*
  - *Mergesort* (1945), *quicksort* (1962)

Sorting algorithms have trade-offs …

- Time (cpu) efficiency
- Space (memory) efficiency
- Data structure (random/sequential access)
- Performance on (mostly) sorted data
- ...

No one algorithm optimizes for all of these!

Evaluating Sorting Algorithms

- In general,
  - \( O(n^2) \) for sorting is “slow”
  - \( O(n \log n) \) for sorting is “fast”
  - \( O(n) \) for sorting is considered “ideal”

- However,
  - Two different \( O(n^2) \) algorithms, e.g., can have very different actual running times
  - This is why “benchmarking” performance on real data is important
  - Also trade-offs in speed (cpu time) vs. space (memory)
Mergesort

Mergesort
[von Neumann, 1945]

• A “divide and conquer” approach
• The basic idea:
  – Divide the list into two halves
  – Sort each half
  – Merge the sorted halves
  … Merging is fast when the two sublists are sorted

Initial list: \( n=4 \)

<table>
<thead>
<tr>
<th>Split</th>
<th>Sort</th>
<th>Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>29 10 14 13</td>
<td>29 10 14 13</td>
<td>10 29 13 14</td>
</tr>
<tr>
<td>29 10 14 13</td>
<td>10 29 13 14</td>
<td>10 13 14 29</td>
</tr>
</tbody>
</table>
Mergesort

- Given that mergesort merges two sorted lists to create a (larger) sorted list
- How do we sort the two sublists?
  - Using mergesort!
  - Divide the list into two halves, sort each half (using mergesort), and then merge the sorted halves

- Mergesort is defined using recursion
Mergesort (based on textbook)

void Mergesort(Entry theArray[], int n, int first, int last)
{
    if(first < last) {
        int mid = (first + last) / 2;
        Mergesort(theArray, n, first, mid);
        Mergesort(theArray, n, mid + 1, last);
        Merge(theArray, n, first, mid, last);
    }
}

Mergesort (based on textbook)

void Merge(Entry theArray[], int n, int first, int mid, int last)
{
    Entry tmpArray[n];
    int first1 = first, first2 = mid + 1, i = first1;
    for(; first1 <= mid && first2 <= last; i++) {
        if(theArray[first1] < theArray[first2])
            tmpArray[i] = theArray[first1++];
        else
            tmpArray[i] = theArray[first2++];
    }
    for(; first1 <= mid; i++, first1++)    // copy remaining from first half
        tmpArray[i] = theArray[first1];
    for(; first2 <= last; i++, first2++)    // copy remaining from sec. half
        tmpArray[i] = theArray[first2];
    for(int j = first; j <= last; j++)    // copy tmpArray to the array
        theArray[j] = tmpArray[j];
}
**Mergesort**

The *merge* step

- Assume number of elements to be merged is $n$
  - $n = \text{length of first half} + \text{length of second half}$

- The merge step requires
  - $n - 1$ comparisons
  - $n$ moves from the original array to the temp array
  - $n$ moves from the temp array to the original array

- So merge costs $3n - 1$

- Which is $O(n)$