Today

- Analysis Examples

Assignments & Announcements

- HW 6 due date moved to Tues
- Quiz on Tues
More Examples: Selection Sort Complexity Analysis

The algorithm (psuedocode):

```plaintext
SelectionSort
Input: array A[], array length n
1. for i = 0 to n-2 // n-1 passes
2. index := i
3. for j = i+1 to n-1 // n-i-1 iterations per pass
5.   index := j
6. swap(A[i], A[index])
```

When comparing sorting algorithms, usually interested in:

- Number of **comparisons** = pairs of array items compared
- Number of **moves** = moving array items (including "swaps")

Recall: Same number of moves on each pass
- since we have \( n - 1 \) passes, \( \Theta(n) \) total moves

Q: What is the cost in terms of comparisons? ... Note: see duplicates

- when \( i = 0 \) \( n-1 \) comparisons
- when \( i = 1 \) \( n-2 \) comparisons
- ...
- when \( i = n-2 \) 1 comparisons

- Which (similar to the duplicates analysis), gives us:

\[
\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n
\]

- And so selection sort is \( \Theta(n^2) \)
More Examples: Merge Sort Complexity Analysis

What is the time complexity of just the merge step?

- Assume number of elements to be merged is \( n \)
  - \( n = \) length of first half + length of second half

- The merge step requires:
  - \( n - 1 \) comparisons (worst case is interleaved values, e.g., \([a, c, e] \ [b, d, f]\))
  - \( n \) moves from original array into the temporary array
  - \( n \) moves from temporary array into original array

- Thus merge is \( \Theta(n) \)
  - since \( n - 1 \) comparisons and \( 2n \) moves
What is the time complexity of mergesort?

- At each “merge phase” we do $\Theta(n)$ amount of work (from merge)
- Given a list of size $n$, how many “merge phases” are there?

$$\text{Assume } n = 16 \quad (= 2^4)$$

Each merge step gives $\frac{1}{2}$ the # of sublists

# of merge steps = # of “doublings” to $n$

If $n$ is a power of 2 ... 
- $n = 2^x$ where $x = \# \text{ of doublings}$
- $\log_2 n = \log_2 2^x = x$ solving for $x$

More generally, $x = \lceil \log_2 n \rceil$

- Thus, $\Theta(\log n)$ merge steps each costing $\Theta(n)$
- This means mergesort is $\Theta(n \log n)$!
Analyzing merge sort using recurrences ...

Given the function:

\[
\text{merge_sort: } A[], \text{ start, end}
\]

1. if start < end // assumed constant cost \( c_1 \)
2. mid = (start + end) / 2 // assumed constant cost \( c_2 \)
3. merge_sort(A, start, mid)
4. merge_sort(A, mid + 1, end)
5. merge(A, start, mid, end) // assumed cost \( c_3 \cdot n \)

assuming we are counting comparisons and moves, and \( n = 2^k \) (to avoid \( \lceil \frac{n}{2} \rceil \))

(1). We have the following recurrence for \( T(n) \)

\[
T(n) = \begin{cases} 
  c_1 & n = 0 \text{ or } n = 1 \\
  c_1 + c_2 + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + c_3 \cdot n + c_4 & n > 1 
\end{cases}
\]

where each \( c_i \) is a constant such that:

- \( c_1 \) is the \texttt{start < end} comparison cost
- \( c_2 \) is the \texttt{mid} calculation cost
- \( c_3 \cdot n + c_4 \) is the \texttt{merge} step cost

(2). We can simplify and generalize the non recurrence terms

\[
T(n) = \begin{cases} 
  \Theta(1) & n = 0 \text{ or } n = 1 \\
  2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 
\end{cases}
\]

(3). From recurrence relations to asymptotic analysis:

- guess and check (e.g., recursion trees, substitution, iteration, etc.)
- solve directly (e.g., characteristic roots)
- master method
The Master Method (for asymptotic bounds)

A “standard recurrence” has the general form:

\[
T(n) = O(1) \text{ for small } n \quad \text{... in merge sort } 0 \leq n \leq 1
\]
\[
T(n) \leq a \cdot T\left(\frac{n}{b}\right) + O(n^d) \quad \text{... in merge sort } a = 2, b = 2, d = 1
\]

where:

- \(a\) is the number of recursive calls
- \(b\) is size of division (in divide and conquer)
- \(\frac{n}{b}\) is assumed to be either \(\lfloor \frac{n}{b} \rfloor\) or \(\lceil \frac{n}{b} \rceil\)
- \(d\) is exponent in the combine step

Note: algorithms exist with \(a \neq b\) (e.g., Karatsuba int multiply has \(a = 3, b = 2\))

If \(T(n)\) is defined by a standard recurrence, then:

\[
T(n) \leq \begin{cases} 
O(n^d \log n) & \text{if } a = b^d \\
O(n^d) & \text{if } a < b^d \\
O(n^{\log_b a}) & \text{if } a > b^d 
\end{cases}
\]

Q: What does this mean for merge sort?

\[
T(n) = \Theta(n \log n)
\]

- because \(d = 1\) and \(a = b^1\)
- and we know the best and worst cases are the same
Quick Sort Complexity Analysis (brief)

In the partition step, what is the worst case?

- when pivot is the smallest (or largest) value
- since we end up with only one partition
- giving one new partition of size $n - 1$

The partition step has $O(n)$ comparisons/swaps

And we do the partition step $n - 1$ times in the worst case!

Therefore, quick sort is $O(n^2)$ in the worst case

However, quick sort is much better in practice ...

- each step in the best case (and on average) partitions into equal-sized halves
- so we have approx. $\log_2 n$ recursive calls (levels) ... $O(\log n)$
- and at each level we have $O(n)$ comparisons and swaps (see above)

Therefore, quick sort is best-case $O(n \log n)$ ... i.e., $\Omega(n \log n)$

- It is also $O(n \log n)$ in the average case ... assuming few “bad” cases
- If random pivot is used, “bad” cases unlikely to occur
**Binary Search**

The basic idea: ... note: for HW-5, must implement iteratively

Iteratively search for a value \( v \) in a non-empty sorted list

1. Pick the **middle item** in a list
2. If \( v < \text{middle item} \), then search in *left half* of list
3. If \( v > \text{middle item} \), then search in *right half* of list
4. If \( v == \text{middle item} \), then found match

Trace the steps to find “C” using binary search:

\[
\begin{array}{ccccccc}
A & B & C & D & E & F & G \\
\end{array}
\]

... Input

\[
\begin{array}{ccccccc}
A & B & C & D & E & F & G \\
\end{array}
\]

... Pick middle item in list

\[
C < D
\]

... New input

\[
\begin{array}{ccccccc}
A & B & C \\
\end{array}
\]

... Pick middle item in list

\[
\begin{array}{ccccccc}
A & B & C \\
\end{array}
\]

C > B

\[
\begin{array}{ccccccc}
C \\
\end{array}
\]

... New input

\[
\begin{array}{ccccccc}
C \\
\end{array}
\]

... Pick middle item in list

\[
\begin{array}{ccccccc}
C \\
\end{array}
\]

C = C

... Found match!
Q: Is binary search faster than linear search in all cases?
   • for arrays, but not for linked lists!
   • ... this is a motivation for binary search trees

Q: What is the best and worst case for binary search?
   • Best: find on first pass
   • Worst: search until one element left (or out of elements)

Q: What is the recurrence relation for binary search?

\[ T(1) = O(1) \quad \text{... either equal or out of elems} \]

\[ T(n) \leq T\left(\frac{n}{2}\right) + O(1) \quad \text{... no recombine, just comparisons} \]

Q: What is the cost using the Master Method? What is \(a, b, d\)?

\[ T(n) \leq O(\log n) \quad \text{... worst case} \]

where:

- \(a = 1\) since we just search left or right (not both)
- \(b = 2\) since we split list in half
- \(d = 0\) since constant number of recombine steps
- thus, \(a = b^d\) (i.e., \(1 = 2^0\)) which gives \(O(n^0 \log n)\)

We can also see this via the recurrence tree (like in merge sort)