SEARCHING AND SORTING
HINT AT ASYMPTOTIC COMPLEXITY

Pinned Piazza note on Supplemental study material. @281. Contains material that may help you study certain topics. It also talks about how to study.
Developing methods

We use Eclipse to show the development of A2 function evaluate. Here are important points to take away from it.

1. If similar code will appear in two or more places, consider writing a method to avoid that duplication.
2. If you introduce a new method, write a specification for it!
3. Before writing a loop, write a loop invariant for it.
4. Have a loop exploit the structure of the data it processes.
5. Don’t expect your first attempt to be perfect. Just as you rewrite and rewrite an essay, we rewrite programs.
Search as in problem set: b is sorted

<table>
<thead>
<tr>
<th>pre: b</th>
<th>post: b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;= v</td>
</tr>
<tr>
<td>?</td>
<td>&gt; v</td>
</tr>
<tr>
<td>b.length</td>
<td>b.length</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>inv: b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>&lt;= v</td>
</tr>
<tr>
<td>?</td>
</tr>
<tr>
<td>&gt; v</td>
</tr>
<tr>
<td>b.length</td>
</tr>
</tbody>
</table>

Methodology:
1. Draw the invariant as a combination of pre and post
2. Develop loop using 4 loopy questions.

```
h = -1; t = b.length;
while (h+1 != t) {
    if (b[h+1] <= v) h = h+1;
    else t = h+1;
}
```

Practice doing this!
Search as in problem set: b is sorted

- **Pre:** b
  - ?

- **Inv:** b
  - ≤ v
  - ?
  - > v

- **Post:** b
  - ≤ v
  - > v

- **h= −1; t= b.length;**

- **while** (h+1 != t) {
  - **if** (b[h+1] <= v) h= h+1;
  - **else** t= h+1;
}

- **b[0] > v?** one iteration.

- **b[b.length-1] ≤ 0?**
- **b.length iterations**
- **Worst case:** time is proportional to size of b

Since b is sorted, can cut ? segment in half. As a dictionary search
Search as in problem set: b is sorted

pre: b

post: b

inv: b

h= –1;  t= b.length;
while (h != t–1) {
    int e= (h + t) / 2;
    // h < e < t
    if (b[e] <= v)  h= e;
    else t= e;
}
Binary search: an $O(\log n)$ algorithm

\[
\begin{array}{c|c|c|c}
0 & h & t \\
\hline
\text{inv: } b & \leq v & ? & > v \\
\end{array}
\]

\[
h = -1; \quad t = b.length;
\]

\[
\text{while} \ (h \neq t-1) \ \{ \\
\quad \text{int } e = (h+t)/2; \\
\quad \text{if } (b[e] \leq v) \ h = e; \\
\quad \text{else } t = e;
\}
\]

Each iteration cuts the size of the ? segment in half.

\[
\begin{array}{c|c|c|c|c|c}
0 & h & e & t \\
\hline
\text{inv: } b & \leq v & ? & ? & > v \\
\end{array}
\]

\[
n = 2**k \ ? \ \text{About k iterations}
\]

Time taken is proportional to $k$, or $\log n$.

\textbf{A logarithmic algorithm}

Write as $O(\log n)$

[explain notation next lecture]
Looking at execution speed

Process an array of size $n$

- $2n+2$, $n+2$, $n$ are all “order $n$” $O(n)$
- Called linear in $n$, proportional to $n$

Number of operations executed

Constant time

Size $n$

- $2n + 2$ ops
- $n + 2$ ops
- $n$ ops
- $n^2$ ops
InsertionSort

A loop that processes elements of an array in increasing order has this invariant:

\[ b[0..i-1] \text{ is sorted} \]

for (int i = 0; i < b.length; i = i + 1) { maintain invariant }
Each iteration, \( i = i + 1 \); How to keep inv true?

<table>
<thead>
<tr>
<th>inv:</th>
<th>b</th>
<th>sorted</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
<td></td>
<td>b.length</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>e.g.</th>
<th>b</th>
<th>2 5 5 5 7</th>
<th>3</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
<td></td>
<td>b.length</td>
<td></td>
</tr>
</tbody>
</table>

Push \( b[i] \) down to its shortest position in \( b[0..i] \), then increase \( i \)

Will take time proportional to the number of swaps needed
What to do in each iteration?

**inv:**

- `b[0..i]` is sorted

**e.g.:**

```plaintext
b = [2, 5, 5, 5, 7] i = 3
```

**Loop body (inv true before and after):**

- Push `b[i]` to its sorted position in `b[0..i]`, then increase `i`
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i = i + 1) {
    Push b[i] down to its sorted position in b[0..i]
}

Many people sort cards this way
Works well when input is nearly sorted

Note English statement in body.
**Abstraction.** Says what to do, not how.
This is the best way to present it. We expect you to present it this way when asked.
Later, show how to implement that with a loop
InsertionSort

// Q: b[0..i-1] is sorted
// Push b[i] down to its sorted position in b[0..i]
int k = i;
while (k > 0 && b[k] < b[k-1]) {
    Swap b[k] and b[k-1]
    k = k–1;
}
// R: b[0..i] is sorted

invariant P: b[0..i] is sorted
except that b[k] may be < b[k-1]
How to write nested loops

// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i = i+1) {
  Push b[i] down to its sorted position in b[0..i]
}

Present algorithm like this

If you are going to show implementation, put in "WHAT IT DOES" as a comment

// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i = i+1) {
  //Push b[i] down to its sorted position in b[0..i]
  int k = i;
  while (k > 0 && b[k] < b[k-1]) {
    swap b[k] and b[k-1];
    k = k-1;
  }
}
InsertionSort

// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i = i+1) {
    Push b[i] down to its sorted position in b[0..i]
}

Pushing b[i] down can take i swaps.
Worst case takes
\[ 1 + 2 + 3 + \ldots + n-1 = (n-1)*n/2 \]
Swaps.

• Worst-case: \(O(n^2)\)
  (reverse-sorted input)
• Best-case: \(O(n)\)
  (sorted input)
• Expected case: \(O(n^2)\)

\(O(f(n))\) : Takes time proportional to \(f(n)\).
Formal definition later

Let \(n = b.length\)
SelectionSort

pre: \( b \) 0 \( \text{b.length} \) ?
post: \( b \) 0 \( \text{b.length} \)

inv: \( b \) 0 \( \text{b.length} \)
\( \text{sorted, } \leq b[i..] \) \( \geq b[0..i-1] \)

Keep invariant true while making progress?

e.g.: \( b \) 1 2 3 4 5 6 9 9 9 7 8 6 9

Increasing \( i \) by 1 keeps inv true only if \( b[i] \) is min of \( b[i..] \)
SelectionSort

Another common way for people to sort cards

Runtime
- Worst-case $O(n^2)$
- Best-case $O(n^2)$
- Expected-case $O(n^2)$

//sort b[], an array of int
// inv: b[0..i-1] sorted AND
// b[0..i-1] <= b[i..]
for (int i = 0; i < b.length; i = i + 1) {
    int m = index of minimum of b[i..];
    Swap b[i] and b[m];
}

sorted, smaller values         larger values
b 0                                    i                                 length

Each iteration, swap min value of this section into b[i]
Swapping b[i] and b[m]

// Swap b[i] and b[m]
int t = b[i];
b[i] = b[m];
b[m] = t;
**Partition algorithm of quicksort**

![Diagram of partition algorithm

Pre:

<table>
<thead>
<tr>
<th>h</th>
<th>h+1</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

Post:

<table>
<thead>
<tr>
<th>h</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= x</td>
<td>x</td>
<td>&gt;= x</td>
</tr>
</tbody>
</table>

x is called the pivot
Not yet sorted
these can be in any order
The 20 could be in the other partition
Not yet sorted
these can be in any order

pivot

20 31 24 19 45 56 4 20 5 72 14 99

partition

19 4 5 14 20 31 24 45 56 20 72 99

j
Partition algorithm

pre: \[ b \begin{array}{|c|c|c|c|c|}
\hline
& h & h+1 & k \\
\hline
b & x & ? & \\
\hline
\end{array} \]

post: \[ b \begin{array}{|c|c|c|c|c|}
\hline
& h & j & k \\
\hline
b & <= x & x & >= x & \\
\hline
\end{array} \]

Combine pre and post to get an invariant

\[ b \begin{array}{|c|c|c|c|c|c|}
\hline
& h & j & t & k \\
\hline
b & <= x & x & ? & >= x & \\
\hline
\end{array} \]
Partition algorithm

Initially, with $j = h$ and $t = k$, this diagram looks like the start diagram.

Terminate when $j = t$, so the “?” segment is empty, so diagram looks like result diagram.

Takes linear time: $O(k + 1 - h)$
QuickSort procedure

/** Sort b[h..k]. */

public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;  // Base case

    int j = partition(b, h, k);
    // We know b[h..j-1] <= b[j] <= b[j+1..k]

    //Sort b[h..j-1] and b[j+1..k]
    QS(b, h, j-1);
    QS(b, j+1, k);
}
QuickSort

Quicksort developed by Sir Tony Hoare (he was knighted by the Queen of England for his contributions to education and CS).

81 years old.

Developed Quicksort in 1958. But he could not explain it to his colleague, so he gave up on it.

Later, he saw a draft of the new language Algol 58 (which became Algol 60). It had recursive procedures. First time in a procedural programming language. “Ah!,” he said. “I know how to write it better now.” 15 minutes later, his colleague also understood it.
Worst case quicksort: pivot always smallest value

| j |
|---|---|---|---|
| x0 | >= x0 | partitioning at depth 0 |
| j |
| x0 | x1 | >= x1 | partitioning at depth 1 |
| j |
| x0 | x1 | x2 | >= x2 | partitioning at depth 2 |

/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;
    int j = partition(b, h, k);
    QS(b, h, j-1);    QS(b, j+1, k);
Best case quicksort: pivot always middle value

Depth 0. 1 segment of size $\sim n$ to partition.

Depth 2. 2 segments of size $\sim n/2$ to partition.

Depth 3. 4 segments of size $\sim n/4$ to partition.

Max depth: about $\log n$. Time to partition on each level: $\sim n$

Total time: $O(n \log n)$.

Average time for Quicksort: $n \log n$. Difficult calculation
QuickSort procedure

/** Sort b[h..k]. */

public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;
    int j = partition(b, h, k);
    // We know b[h..j-1] <= b[j] <= b[j+1..k]
    // Sort b[h..j-1] and b[j+1..k]
    QS(b, h, j-1);
    QS(b, j+1, k);
}

Worst-case: quadratic
Average-case: O(n log n)

Worst-case space: O(n*n)! --depth of recursion can be n
Can rewrite it to have space O(log n)
Average-case: O(n * log n)
Partition algorithm

Key issue:
How to choose a pivot?

Choosing pivot
- Ideal pivot: the median, since it splits array in half
But computing median of unsorted array is $O(n)$, quite complicated

Popular heuristics: Use
- first array value (not good)
- middle array value
- median of first, middle, last, values GOOD!
- Choose a random element
Quick sort with logarithmic space

Problem is that if the pivot value is always the smallest (or always the largest), the depth of recursion is the size of the array to sort.

Eliminate this problem by doing some of it iteratively and some recursively
Quicksort with logarithmic space

Problem is that if the pivot value is always the smallest (or always the largest), the depth of recursion is the size of the array to sort.

Eliminate this problem by doing some of it iteratively and some recursively. We may show you this later. Not today!
/** Sort b[h..k]. */

public static void QS(int[] b, int h, int k) {
    int h1 = h; int k1 = k;
    // invariant b[h..k] is sorted if b[h1..k1] is sorted
    while (b[h1..k1] has more than 1 element) {
        Reduce the size of b[h1..k1], keeping inv true
    }
}
QuickSort with logarithmic space

```java
/** Sort b[h..k]. */

public static void QS(int[] b, int h, int k) {
    int h1 = h; int k1 = k;
    // invariant b[h..k] is sorted if b[h1..k1] is sorted
    while (b[h1..k1] has more than 1 element) {
        int j = partition(b, h1, k1);
        // b[h1..j-1] <= b[j] <= b[j+1..k1]
        if (b[h1..j-1] smaller than b[j+1..k1])
            { QS(b, h, j-1); h1 = j+1; }
        else
            { QS(b, j+1, k1); k1 = j-1; }
    }
}
```

Only the smaller segment is sorted recursively. If b[h1..k1] has size n, the smaller segment has size < n/2. Therefore, depth of recursion is at most log n.
Binary search: find position $h$ of $v = 5$

**pre:** array is sorted

$h = -1$

```
1 4 4 5 6 6 8 8 10 11 12
```

$t = 11$

$h = -1$

```
1 4 4 5 6 6 8 8 10 11 12
```

$t = 5$

$h = 2$

```
1 4 4 5 6 6 8 8 10 11 12
```

$h = 3$

```
1 4 4 5 6 6 8 8 10 11 12
```

$h = 3$

```
1 4 5 6 6 8 8 10 11 12
```

$t = 4$

**Loop invariant:**

- $b[0..h] \leq v$
- $b[t..] > v$
- B is sorted

**post:** $h \\ h > v$