Supplemental Document for Reading #5

Analyzing Running Time as a Function of Input Size

How do you measure running time as a function of the ***size*** of the input? This has, arguably, been done informally throughout these notes; this is discussed a bit more formally and extensively in this document.

Two Example Algorithms

Consider the following problem.

Searching in an Array

Precondition: An integer array `A` and an integer `key` are given as input.

Postcondition: If `A[i]` $=$ `key` for an integer `i` such that $0 \leq$ `i` $<$ `A.length` then the *smallest* such integer is returned as output. An `ElementNotFoundException` is thrown otherwise.

The following algorithm can be used to solve this problem.

```
integer search(integer[] A, integer key) {
1. integer i := 0
2. while ((i < A.length) and (A[i] ≠ key)) {
3.   i := i + 1
   }
4. if (i < A.length) {
5.   return i
   } else {
6.   throw an ElementNotFoundException
   }
}
```

Before considering its running time, let us prove the ***correctness*** of this algorithm.

- If the precondition for the "Searching in an Array" problem is satisfied then the following assertion is satisfied immediately before the step at line 1:

  Assertion:
  1. `A` is an input integer array.
  2. `key` is an integer input.

- The following ***loop invariant*** for the `while` loop in this algorithm can be established:

  Loop Invariant:
  1. `A` is an input integer array.
  2. `key` is an integer input.
  3. `i` is an integer variable such that $0 \leq$ `i` $\leq$ `A.length`.
  4. `A[j]` $\neq$ `key` for every integer `j` such that $0 \leq$ `j` $<$ `i`.

- Assertions before the loop, at the beginning of the loop body, at the end of the loop body, and immediately after the loop, are as follows.

  Assertion Before the Loop:
  1. The loop invariant is satisfied.
  2. `i` $= 0$.

  Assertion at the Beginning of the Loop Body:
  1. The loop invariant is satisfied.
  2. $0 \leq$ `i` $<$ `A.length`.
  3. `A[i]` $\neq$ `key`.

  Assertion at the End of the Loop Body: The loop invariant is satisfied.

  Assertion Immediately After the Loop:
  1. The loop invariant is satisfied.
  2. Either `i` $=$ `A.length`, or $0 \leq$ `i` $<$ `A.length` and `A[i]` $=$ `key`.

  Exercise: Use the techniques (or modify example proofs) from Reading #3 to establish that all of these assertions are correct.

- Since the last of these assertions holds immediately before the test at line 4, an examination of the `if-then-else` test at lines 4–6 confirms that — if this test is reached at all,

- If `key` is an entry of the array `A`, then the smallest integer `i` such that $0 \leq$ `i` $<$ `A.length` and `A[i]` $=$ `key` is returned, as output, at line 5; and
- if `key` is *not* an entry of the array `A`, then an `ElementNotFoundException` is thrown at line 6.

Consequently, the postcondition for the "Searching in an Array" problem is satisfied if the execution of the algorithm ends — *if* the execution ends at all. It is also easily checked that this algorithm has no undocumented side-effects.

In other words, this algorithm is ***partially correct.***

- In order to show that an execution of the algorithm (beginning with the precondition for the "Searching in an Array" problem satisfied) *does* end, one can show that the function

$$f(\texttt{A}, \texttt{i}) = \texttt{A.length} - \texttt{i}$$

is a ***bound function*** for the `while` loop in this algorithm.

Exercise: Use the above, along with techniques introduced in Reading #4, to to prove that if the algorithm is executed when the precondition for the "Searching in an Array" problem is satisfied, then this execution of the algorithm is guaranteed to ***terminate***.

Note: It now follows that this algorithm correctly solves the "Searching in an Array" problem.

Let's assume that the integers that are stored in the array (and used as the key) are reasonably small — so that the uniform cost criterion can (reasonably) used to define the running time of the algorithm, and the ***size*** of its input.

- We will define the ***size*** of the array `A` to be its length, `A.length`.
- Since the ***size*** of the integer `key` is 1 it would be quite reasonable to define the ***input size*** for this problem (and algorithm) to be `A.length` $+ 1$.
- However, it is common practice to state the cost of a "search" algorithm as a function of the length of the array it searches in. Thus (to be consistent with that) we will overlook the size of the `key` and define the ***size*** of the input for this problem to be `A.length`.

Now consider the following *recursive* algorithm. This accesses (but does not modify) an integer array `A` and an integer `key` as global data. It receives a pair of integers, `first` and `last`, such that $0 \leq$ `first` $\leq$ `A.length` and $-1 \leq$ `last` $\leq$ `A.length` $- 1$ as inputs.

```
integer recSearch(integer first, last){
1.  if (first > last) {
2.    throw an ElementNotFoundException
3.  } else if (A[first] == key) {
4.    return first
    } else {
5.    return recSearch(first+1, last)
    }
}
```

Exercises

1. Prove that the value $\texttt{last}-\texttt{first}+1$ is a ***bound function for this recursive algorithm.***

2. Prove the following claim. (The case $\texttt{first} > \texttt{last}$ can be dealt with by inspecting the code and discussing what happens. The case $\texttt{first} \leq \texttt{last}$ can be dealt with using induction on $\texttt{last} - \texttt{first}$.)

   Claim: Suppose that algorithm `recSearch` accesses an integer array `A` and an integer `key` as global data.

   If this algorithm is executed on integer inputs `first` and `last` such that $0 \leq \texttt{first} \leq \texttt{A.length}$ and such that $-1 \leq \texttt{last} \leq \texttt{A.length} - 1$, then the following properties are satisfied:

   (a) If $\texttt{first} > \texttt{last}$ then the execution of the algorithm eventually halts, and an `ElementNotFoundException` is thrown when it does.

   (b) If $\texttt{first} \leq \texttt{last}$, so that $0 \leq \texttt{first} \leq \texttt{last} \leq \texttt{A.length} - 1$, and there exists an integer `i` such that $\texttt{first} \leq \texttt{i} \leq \texttt{last}$ and $\texttt{A[i]} = \texttt{key}$, then the execution of the algorithm eventually halts, and the *smallest* integer `i` satisfying the above conditions is returned as output.

   (c) If $\texttt{first} \leq \texttt{last}$, so that $0 \leq \texttt{first} \leq \texttt{last} \leq \texttt{A.length} - 1$, and $\texttt{A[i]} \neq \texttt{key}$ for every integer `i` such that $\texttt{first} \leq \texttt{i} \leq \texttt{last}$, then the algorithm eventually halts, and an `ElementNotFoundException` is thrown when it does.

This algorithm has no undocumented side-effects. It now follows that the "Searching in an Array" Problem is *also* correctly solved by the algorithm that follows.

```
integer search2 (integer[] A, integer key) {
1.   return recSearch(0, A.length−1)
}
```

As for the previous `search` algorithm, we will define the ***size*** of the input to be the length `A.length` of the input array.

Question: How can we define the running time of *these* algorithms as functions of the ***size*** of their input? The running times also depend on whether the `key` is found in the array, and on *where* it is found, if it is.

Worst-Case Running Time

Definition: The ***worst-case running time*** of an algorithm on a nonnegative integer input size `n` is the **maximum** of all of the running times for executions of this algorithm on "valid" inputs (that is, inputs satisfying the precondition for the problem to be solved) with **size** `n`.

- Note that this is a function of the **input size** `n`. It is undefined, at `n`, if there are no valid inputs with size `n` at all.
- This will be the measure of running time, as a function of input size, that is considered most often in this course, and in other courses that you will take after this.
- *Generalization:* If an algorithm accesses and modifies global data then running time can be defined as a function of the **size** of the input *and* the global data that is accessed. (We will not need to do this today.)

Finding and Proving Upper Bounds for Worst-Case Running Time

First Objective: Finding a function $f_H(\mathtt{n})$ such that (whenever it is defined) the worst-case running time $T_A(\mathtt{n})$ of a given algorithm A for input size `n` is **less than or equal to** $f_H(n)$.

- If the above condition is satisfied then the function f_H is an ***upper bound*** for the worst-case running time of the algorithm A.

Note: In general, we will **not** be trying to prove that the worst-case running time is actually *equal to* f_H — but *sometimes* we will be able to do that too.

Recall that the function $f(\mathtt{A}, \mathtt{i}) = \mathtt{A.length} - \mathtt{i}$ is a **bound function** for the `while` loop in the `search` algorithm.

- Note that, since the **input size** for this algorithm is `A.length`, this is actually of a function of the input size and of the value of a local variable (namely, `i`).

- Since the initial value of the variable `i` is 0, the initial value of this function is `A.length`.
- As discussed in Reading #4, it follows that there are **at most** `A.length` executions of the body of the `while` loop, included in the execution of the loop, when this algorithm is executed (with an array `A` included as part of the input).
- Suppose that the loop body is executed exactly h times — so that the loop test is executed exactly $h + 1$ times. Then
  - For every integer j such that $1 \leq j \leq h+1$, the cost $T_{\textit{test}}(j)$ of the j^{th} of the `while` loop's *test* is 1.
  - Since the loop body consists of a single statement, the cost $T_{\textit{body}}(j)$ of the j^{th} execution of the loop *body* is 1, as well, for every integer j such that $1 \leq j \leq h$.
  - Adding everything up, we see that the number of steps included in the execution of the loop (in this case) is

$$\sum_{j=1}^{h+1} T_{\textit{test}}(j) + \sum_{j=1}^{h} T_{\textit{body}}(j) = \sum_{j=1}^{h+1} 1 + \sum_{j=1}^{h} 1 = 2h + 1.$$

  As noted above, $h \leq$ `A.length`, so the number of steps used during an execution of the `while` loop is at most $2 \times$ `A.length` $+ 1$.
- Exactly *one* step (at line 1) is executed before the loop, during an execution of this algorithm — and exactly *two* steps (either at lines 4 and 5, or at lines 4 and 6) are executed after it. Thus the number of steps included in an execution of the algorithm is *at most* $2 \times$ `A.length` $+ 4$.

  Now recall that the ***input size*** was defined to be `A.length`...

Conclusion: The worst-case running time $T_{\texttt{search}}(\texttt{n})$, of the `search` algorithm, for a nonnegative input size `n`, is **less than or equal to** $2\texttt{n} + 4$.

Method Used Here: *Same as for Reading #4.* If the **bound function** for a `while` loop that you establish depends on the **input size** instead of the **input value** then this will often be sufficient to prove an upper bound for worst-case running time.

Next recall that `search2` calls the `resSearch` algorithm.

- Notice that if it is called with integer inputs `first` and `last`, then `recSearch` is searching for the integer `key` in a part of the array `A` that includes $k = \max(\texttt{last} - \texttt{first} + 1, 0)$ entries.

- With that noted, for an integer $k \geq 0$, let us define $T_{\texttt{recSearch}}(k)$ to be the **maximum** number of steps carried out by the algorithm `recSearch` when it is executed using an integer array `A` and an integer `key` as global data and is called with integer inputs `first` and `last` such that
  - $0 \leq \texttt{first} \leq \texttt{A.length}$,
  - $-1 \leq \texttt{last} \leq \texttt{A.length} - 1$, and
  - $\max(\texttt{last} - \texttt{first} + 1, 0) = k$.

  This will be what we need to find an upper bound for the worst-case running time of `search2`.

- *Case:* $k = 0$.
  - In this case it follows by the definition of k that $\texttt{first} \geq \texttt{last} + 1$, so the test at line 1 of the algorithm is passed, and that exactly two steps — at lines 1 and 2 — are included in this execution of the algorithm. Thus

$$T_{\texttt{recSearch}}(0) = 2.$$

- *Case:* $k \geq 1$.
  - In this case $\texttt{first} \leq \texttt{last}$ — so that $0 \leq \texttt{first} \leq \texttt{last} \leq \texttt{A.index} - 1$ — and the test at line 1 fails.
  - The execution of the algorithm continues with the test at line 3. Of course, this test either passes or fails. These "subcases" are considered separately, below.
    - *Subcase:* The test at line 3 passes. In this case the execution of the algorithm ends with one more step — at line 4. In total, 3 steps have been executed.
    - *Subcase:* The test at line 3 fails. In this case the execution of the algorithm continues with the step at line 5 — which initiates a recursive application of the algorithm with inputs $\texttt{first} + 1$ and `last`. The number of steps used by this recursive application is — by definition — **at most** $T_{\texttt{recSearch}}(k-1)$ so that, in total, this execution of the algorithm has included **at most** $3 + T_{\texttt{recSearch}}(k-1)$ steps.
  - *Conclusion:* Since the maximum possible running time is at most the **maximum** of the bounds obtained for the above subcases,

$$T_{\texttt{recSearch}}(k) \leq \max(3, 3 + T_{\texttt{recSearch}}(k-1)).$$

- The following ***recurrence*** giving an upper bound for $T_{\texttt{recSearch}}(k)$ has now been obtained:

$$T_{\texttt{recSearch}}(k) \leq \begin{cases} 2 & \text{if } k = 0, \\ \max(3, 3 + T_{\texttt{recSearch}}(k-1)) & \text{if } k \geq 1. \end{cases}$$

- ***Exercise:*** Use the above recurrence to prove (by induction on k) that

$$T_{\texttt{recSearch}}(k) \leq 3k + 2$$

for every integer $k \geq 0$.

Now consider the `search2` algorithm.

- Note that — when given an integer array `A` and an integer `key` are given as inputs — the algorithm `search2` executes a single step, in which it calls the algorithm `recSearch` with inputs 0 and `A.length` $- 1$... corresponding to the value $k =$ `A.length`.
- Thus, when executed with an array `A` as input, `search2` uses one more step than `recSearch` does when the latter algorithm is called with inputs such that $k =$ `A.length`.
- Recall that the **input size** for `search2` has been defined to be `A.length`.

Conclusion: The worst-case running time $T_{\texttt{search2}}(\texttt{n})$, of the `search2` algorithm, for a non-negative input size `n`, is **less than or equal to** $3\texttt{n} + 3$.

Method Used Here: *Same as for Reading #5.* If you can find a ***recurrence*** giving an upper bound for the running time for a recursive algorithm, as a function of the **input size** instead of the **input value**, then solving this recurrence gives an upper bound for the worst-case running time of the algorithm.

Do Not Do This Instead!

Sometime, on an an assignment or test, students will write something like the following. This is given, below, as a solution for the problem, "Find an upper bound for the worst-case running time of the `search` algorithm."

> The ***worst-case input*** includes an array `A` and `key` such that the `key` is not in the array.
>
> The number of steps executed in this case is $2\texttt{n} + 4$ if `A` has length `n`.
>
> Therefore the worst-case running time $T_{\texttt{n}}(\texttt{n})$ is (less than or equal to) $2\texttt{n} + 4$.

Question: What comment might a marker provide... and what mark will probably be awarded?

Answer: The *comment* written might be

This is not a Proof.

and the *mark* awarded will — very possibly — be 0.

Of course, you might be wondering *why that is the case.* After all, the number of steps executed really *is* $2\mathtt{n} + 4$ if the `key` is not in the input array `A` and this array has length `n`.

Hint: If the marker wanted to give better feedback then she or he might circle or underline the phrase, "The worst-case input is" at the beginning of the student's answer, and then write the comment

How can you **Prove** This?

Here's the Problem:

- In general, identifying one particular input, with a given size, whose running time really ***is*** probably greater than or equal to those of all the *other* inputs with a given size, can be ***very*** hard to do — and the student's answer ***does not*** include the kind of proof that is needed.
- Instead, the student has effectively "dumbed-down" the problem — replacing it with a different problem that is much easier to solve.
- *In particular:* Instead of proving that the running times **for all** inputs with a given size are less than or equal to a given value (proving a ***universal*** claim), the student has pulled one input out of a hat and analyzed *its* running time — which would only be allowed if you were trying to prove an ***existential*** claim, instead.

Finding and Proving Lower Bounds for Worst-Case Running Time

Suppose, again, that $T_{\mathtt{A}}(\mathtt{n})$ is the worst case running time of an algorithm `A` when executed on an input with size `n` (for $\mathtt{n} \geq 1$).

Definition: A value B is a ***lower bound*** for $T_{\mathtt{A}}(\mathtt{n})$ if $T_{\mathtt{A}}(\mathtt{n}) \geq B$, and a function $g(\mathtt{n})$ is a ***lower bound*** for $T_{\mathtt{A}}$ if $T_{\mathtt{A}}(\mathtt{n}) \geq g(\mathtt{n})$ for every nonnegative integer `n` for which $T_{\mathtt{A}}(\mathtt{n})$ is defined.

Proving a Lower Bound: In order to show that $T_{\mathtt{A}}(\mathtt{n}) \geq B$, for a given value B, we must show that **there exists** an input `I` (that satisfies the precondition of the problem that algorithm `A` solves) with size `n` such that the running time of `A` on input `I` is greater than or equal to B.

Thus, we must prove an **existentially quantified statement** when establishing a lower bound for the worst-case running time of an algorithm.

With that noted, let us consider the `search` algorithm introduced at the beginning of these notes.

- ***Complication:*** When finding a lower bound for the worst-case running time of an algorithm with a `while` loop, you generally need to find a lower bound for the number of the executions of the body of the loop, when the algorithm is executed on some (carefully chosen) input.
- This is *not* something that a **bound function** for the `while` loop can be used to provide (although that might *suggest* something).
- It will probably be necessary to state and prove some additional information about what happens during the execution of the `while` loop, on this input, in order to do this. *Mathematical induction* can often be used to prove the information that is needed.
- With that noted, let `n` be a nonnegative integer. Consider an array `A` with length `n` and and integer `key` such that `A[i]` $\neq$ `key` for every integer `i` such that $0 \leq$ `i` $<$ `n`. Note that these form an input satisfying the precondition for the "Searching in an Array" problem with input size `n`. Consider the following.
- ***Claim:*** Suppose that `n` is a nonnegative integer, `A` is an integer array with length `n`, and that `key` is an integer such that `A[i]` $\neq$ `key` for every integer `i` such that $0 \leq$ `i` $<$ `n`. Consider an execution of the `search` algorithm with `A` and `key` as inputs.

  For every integer `h` such that $1 \leq h \leq$ `n`, there are at least h executions of the body of the loop, during the execution of the `while` loop as part of this execution of the algorithm. Furthermore, the variable `i` has value $h - 1$ at the beginning of the h^{th} execution of the loop body, and this variable has value h at the *end* of this execution of the loop body.
- ***Exercise:*** After reviewing the notes from Reading #1, as needed, prove the above claim — or, at least, make sure that you understand how to do this.

  Note how the information about the value of `i`, that is included in the claim, makes it easier to use induction to prove it!
- Now consider an execution of the `search` algorithm that includes exactly ℓ executions of the body of the loop, for some nonnegative integer ℓ.
  - One can see by an examination of the code that each execution of the body of the loop includes exactly one step (namely, an execution of the step at line 3).

  - Since the loop body does not include any `break`, `exit` or `return` statements, there must always be exactly one more execution of the loop *test* than executions of the loop body. Thus there are exactly $\ell + 1$ execution of the loop tests included in this execution of the `search` algorithm.

- Each execution of the loop test includes one step, namely, the execution of the step at line 2.

- The number of steps included in this execution of the loop is, therefore,

$$\sum_{j=1}^{\ell} 1 + \sum_{j=1}^{\ell+1} 1 = \ell + (\ell + 1) = 2\ell + 1.$$

- ***Conclusion:*** Since the number of execution of the loop body for the input being considered is $\ell \geq \mathtt{n}$, it follows that the number of steps included in the execution of the loop, for this input, is **greater than or equal to** $2\mathtt{n} + 1$.

- As previously noted, every execution of this algorithm includes exactly *one* step *before* the loop (at line 1) and exactly *two* steps (either at lines 4 and 5, or at lines 4 and 6) *after* it.

  The number of steps included in an execution of the `search` algorithm on the input being considered is, therefore, **greater than or equal to** $2\mathtt{n} + 4$.

Conclusion: The worst-case running time $T_{\mathtt{search}}(\mathtt{n})$, of the `search` algorithm, for a nonnegative input size `n`, is **greater than or equal to** $2\mathtt{n} + 4$.

Note: Since the upper and lower bounds proved for this match, we have actually **proved** that

$$T_{\mathtt{search}}(\mathtt{n}) = 2\mathtt{n} + 4$$

for every nonnegative integer `n`.

Next consider the recursive search algorithm, `recSearch`, and the algorithm `search2` that calls it.

- ***Complication:*** When a *recursive* algorithm is executed on a given input, the execution includes executions of the *same* algorithm on other inputs (with different sizes) too.

- Consequently, when computing a lower bound for the running time of a recursive algorithm, you typically need to find lower bounds for *all* of the executions of the running times of the algorithm on some **set** S of inputs,

  - that includes the execution whose running time you wish to bound, and

– such that if the recursive algorithm is executed on *one* input in this set, and this results in a recursive application of the algorithm, then the input for the recursive application belongs to this set too.

- Once again, a **recurrence** can (at least, possibly) be formed and solved to prove a lower bound in this case.

- With that noted, the following can be proved by induction on k:

  Claim: Let `n` be a positive integer, let `A` be an integer array with length `n`, and let `key` be an integer such that `A[i]` $\neq$ `key` for every integer `i` such that $0 \leq$ `i` $<$ `n`.

  Then the following property is satisfied for every nonnegative integer k:

  > If `recSearch` is executed with integer inputs `first` and `last` such that $0 \leq$ `first` $\leq$ `n`, $-1 \leq$ `last` $\leq$ `n` $- 1$, and $\max($`last` $-$ `first` $+ 1, 0) = k$, using the above array `A` and integer `key` as global data, then this execution of the algorithm halts after exactly $\widetilde{T}_{\texttt{recSearch}}(k)$ steps, where

$$\widetilde{T}_{\texttt{recSearch}}(k) = \begin{cases} 2 & \text{if } k = 0, \\ 3 + \widetilde{T}_{\texttt{recSearch}}(k-1) & \text{if } k \geq 1. \end{cases}$$

  Proof: Let `n`, `A` and `key` be as in the statement of the claim. It can be established that the property in the claim is satisfied, for all integers k such that $0 \leq k \leq$ `n`, using induction on k. The standard form of mathematical induction will be used.

  Basis: Suppose first that $k = 0$; then it is necessary and sufficient to show that if the `recSearch` algorithm is executed with integer inputs `first` and `last` such that $0 \leq$ `first` $\leq$ `n`, $-1 \leq$ `last` $\leq$ `n` $- 1$, and $\max($`last` $-$ `first` $+ 1, 0) = 0$, using the above array `A` and integer `key` as global data, then this execution of the algorithm halts after $\widetilde{T}_{\texttt{recSearch}}(0) = 2$ steps.

  With that noted, consider any such execution of this algorithm. Since

$$\max(\texttt{last} - \texttt{first} + 1, 0) = 0,$$

  `last` $-$ `first` $+ 1 \leq 0$, so that `first` $>$ `last`.

  One can see from this, by inspection of the code, that this execution includes the test at line 1, which passes, and an execution of the step at line 2 — after which the algorithm halts.

  Since two steps have been taken, this establishes the desired property in this case.

  Inductive Step: Let h be an integer such that $h \geq 0$. It is necessary and sufficient to use the following

Inductive Hypothesis: If the recSearch algorithm is executed with integer inputs first and last such that $0 \leq \texttt{first} \leq \texttt{n}$, $-1 \leq \texttt{last} \leq \texttt{n} - 1$, and such that

$$\max(\texttt{last} - \texttt{first} + 1, 0) = h,$$

using the above array A and integer key as global data, then this execution of the algorithm of the halts after exactly $\widetilde{T}_{\texttt{recSearch}}(h)$ steps, where the function $\widetilde{T}_{\texttt{recSearch}}$ as defined in the claim.

in order to define the following

Inductive Claim: If recSearch is executed with integer inputs first and last such that $0 \leq \texttt{first} \leq \texttt{n}$, $-1 \leq \texttt{last} \leq \texttt{n} - 1$, and such that

$$\max(\texttt{last} - \texttt{first} + 1, 0) = h + 1,$$

using the above array A and integer key as global data, then this execution of the algorithm of the halts after exactly $\widetilde{T}_{\texttt{recSearch}}(h + 1)$ steps, where the function $\widetilde{T}_{\texttt{recSearch}}$ as defined in the claim.

With that noted, consider an execution of the recSearch algorithm on inputs first and last such that $0 \leq \texttt{first} \leq \texttt{n}$, $-1 \leq \texttt{last} \leq \texttt{n} - 1$, and

$$\max(\texttt{last} - \texttt{first} + 1, 0) = h + 1,$$

using the above array A and integer key as global data.

Since $h + 1 \geq 1$ and $\max(\texttt{last} - \texttt{first} + 1, 0) = h + 1$, $\texttt{last} - \texttt{first} + 1 = h + 1$, so that $\texttt{last} = \texttt{first} + h \geq \texttt{first}$. The execution of the algorithm therefore begins with the execution of the test at line 1 — which fails, so that the execution of the algorithm continues with the test at line 3.

Now, since $0 \leq \texttt{first} \leq \texttt{n}$ and $-1 \leq \texttt{last} \leq \texttt{n} - 1$ it follows (since $\texttt{first} \leq \texttt{last}$, as noted above), that

$$0 \leq \texttt{first} \leq \texttt{last} \leq \texttt{n} - 1 = \texttt{A.length} - 1.$$

It is given that A[i] $\neq$ key for every integer i such that $0 \leq \texttt{i} \leq \texttt{n} - 1$ — so that, in particular, A[first] $\neq$ key, and the test at line 3 fails as well. The execution of the algorithm therefore continues with the step at line 5.

The number of steps included in the execution of the step at line 5 is one more than the number of step included in the recursive application of the algorithm started at that line. One can see by inspection of the code that the inputs for this execution are the integers $\texttt{first}' = \texttt{first} + 1$ and $\texttt{last}' = \texttt{last}$.

- Since $0 \leq \texttt{first} \leq \texttt{n} - 1$ (as noted above), $0 \leq \texttt{first}' \leq \texttt{n}$.

– Since $0 \leq \texttt{last} \leq \texttt{n} - 1$ (as noted above), $0 \leq \texttt{last}' \leq \texttt{n} - 1$.
– Since $\texttt{first}' = \texttt{first} + 1$ and $\texttt{last}' = \texttt{last}$,

$$\begin{aligned}(\texttt{last}' - \texttt{first}' + 1) &= (\texttt{last} - (\texttt{first} + 1) + 1) \\ &= (\texttt{last} - \texttt{first} + 1) - 1 \\ &= (h + 1) - 1 = h.\end{aligned}$$

Since the same global data (the array A and integer key) are used in this recursive application as the one before it, it now follows the Inductive Hypothesis that the number of steps included in the recursive application at line 5 is $\widetilde{T}_{\texttt{recSearch}}(h)$. This implies that the total number of steps included in the execution of the algorithm with inputs first and last is exactly $3 + \widetilde{T}_{\texttt{recSearch}}(h)$.

Now, since $h + 1 \geq 1$, one can see by the definition of this function in the claim that this number of steps is equal to $\widetilde{T}_{\texttt{recSearch}}(h + 1)$ — as needed to complete the inductive step and the proof of the claim. □

- ***Exercise:*** Use the recurrence (from the previous claim)

$$\widetilde{k}_{\texttt{recSearch}}(k) = \begin{cases} 2 & \text{if } k = 0, \\ 3 + \widetilde{T}_{\texttt{recSearch}}(k-1) & \text{if } k \geq 1, \end{cases}$$

to ***prove*** that $\widetilde{T}_{\texttt{recSearch}}(k) = 3k + 2$ for every nonnegative integer k.

- Notice next that (since $T_{\texttt{recSearch}}(k)$ represents worst-case running time)

$$\widetilde{T}_{\texttt{recSearch}}(k) \leq T_{\texttt{recSearch}}(k)$$

for every nonnegative integer k, for the function $T_{\texttt{research}}$ considered earlier in these notes. The above information, and the upper bound for this function already proved, now imply that

$$T_{\texttt{recSearch}}(k) = 3k + 2$$

for every nonnegative integer k, as well.

- Note next that the worst-case running time $T_{\texttt{search2}}(\texttt{n})$ of the search2 algorithm is *greater than or equal to* the number of steps used by this algorithm when it is executed with an array A of length n, and an integer key.

Indeed, an examination of the search2 algorithm should confirm that the number of steps taken by this algorithm, on this input, is $\widetilde{T}_{\texttt{recSearch}}(\texttt{n}) + 1 = 3\texttt{n} + 3$.

Conclusion: $T_{\texttt{search2}}(\texttt{n}) \geq 3\texttt{n} + 3$ — and, together with the matching *upper bound* that was already proved — this implies that

$$T_{\texttt{search2}}(\texttt{n}) = 3\texttt{n} + 3$$

for every integer $\texttt{n} \geq 0$.

It is OK if Upper Bounds and Lower Bounds Do Not Agree.

Indeed, that is the more usual situation when you are analyzing algorithms that are more complicated than the ones in these examples.

Worst-Case Running Time: A Strength and a Weakness

Strength: The worst-case running time of an algorithm for a given input size really does (by definition) given an upper bound for the running time used when the algorithm is executed on a valid input with that size — provided that there were no mistakes in its calculation (so that it really *is* the worst-case running time), you can rely on that!

Weakness: Sometimes the worst-case running time can be very *pessimistic* in the following sense: The running time for *many* or even *most* valid inputs with a given size might be much, much **smaller**. It might be extremely unlikely that the "worst-case running time" is actually required very often (if at all)!

Other Functions of Running Time, as Functions of Input Size

- The ***best-case running time*** of an algorithm on a nonnegative integer input size $\texttt{n}$ is the **minimum** of all the running times for executions of this algorithm on "valid" inputs (that is, inputs satisfying the precondition for the problem to be solved) with **size** $\texttt{n}$.

  This is rarely of independent interest. However, if it can be shown that the worst-case running time of an algorithm and the best-case running time of the algorithm (for the same input size) are (almost) the same, then it follows that the worst-case running time is *not* an overly pessimistic measure.

  This is *not* true for the example algorithms given above: By considering input arrays $\texttt{A}$ and an integer $\texttt{key}$ such that $\texttt{key} = \texttt{A}[0]$, one can confirm that the best case running times of each of these algorithms is a small positive constant — not dependent on the input size n at all!

- It is also possible to apply material from probability theory to define the ***expected running time***, or ***average-case running time*** of an algorithm.

  However, this generally requires that you make additional assumptions that might not be valid: The set of all valid inputs of a given size $\texttt{n}$ must be used to define a *sample*

space. A *probability distribution* for this sample space must, somehow, be defined — which generally requires additional assumptions — and the expected running time can then be defined as the *expected value of a random variable*.

This was — possibly — used in CPSC 331 or 319 when making claims about the expected behaviour of the `QuickSort` algorithm.