Chapter 1: Analyzing Algorithms and Problems

JAVA Conventions for Describing Algorithms

Since algorithms are not programs, you can use pretty much any modern computer language to describe your algorithms. We choose to use JAVA-like notation since our introduction courses are taught in JAVA.

JAVA Conventions

Here is the list of conventions that will be used.

- Use either braces ("{" and "}") or indentation to indicate blocks.
- Omission of keywords, visibility control names (**public**, **private**, and **protected**) and **static**.
- Class name qualifiers are omitted from method calls.
- Logical operators: $==$, $\neq$, $\leq$, $\geq$, $<$, $>$, &&, and $||$
- Mathematical operators: $+$, $*$, $-$, $/$, $++$, $--$, $+=$, $*=$, $-=$, and $/=$
- Control strings: **if–else**, **for**, **while**, **return**, and **break**.

Mathematical Background

Set notation

- $\in$ for "is an element of" and $\subseteq$ for "is a subset of"
- Set operations: $\bar{}$ (complement), $\cap$ (intersection), $\cup$ (union), $\setminus$ (subtraction)

Binomial coefficients, $C(n,k)$ and $\binom{n}{k}$

Here n and k are nonnegative integers such that $0 \leq k \leq n$. Both $C(n,k)$ and $\binom{n}{k}$ denote the number of ways to select k things from n things. For all n, $C(n,0) = 1$.

Cross Products and Binary Relations

- For sets A and B, $A \times B$ is the set $\{(x, y) \mid x \in A \wedge y \in B\}$.
- A simple cross product is one of the form $A \times A$. A binary relation is a subset of a simple Cartesian product. A binary relation $R \subseteq A \times A$ is
  - **reflexive** if for all x in A, $(x, x) \in A$,
  - **symmetric** if for all x and y in A, $(x, y) \in A$ if and only if (y, x),
  - **antisymmetric** if for all x and y in A, at most one of (x, y) and (y, x) is a member of A
  - **transitive** if for all x, y, and z in A, $(x, y) \in A \wedge (y, z) \in A$ implies $(x, z) \in A$.

Mathematical Functions

- The ceiling function: $\lceil x \rceil$ is the smallest integer not smaller than x.
- The floor function: $\lfloor x \rfloor$ is the largest integer not exceeding x.
- For all positive reals x and b, $\log_b x$ is the logarithm of x base x, that is, the real y such that $b^y = x$.

  Use lg to denote $\log_2$ and ln to denote $\log_e$.

  Lemma 1.1 (page 15) presents some fundamental properties of logarithms.

Summations and Series

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.
- $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$.
- For all $k \geq 0$, $\sum_{i=1}^{n} i^k \approx \frac{1}{k+1} n^{k+1}$.
- **Geometric Series** For all $r \neq 1$, $\sum_{i=0}^{n} r^i \approx \frac{r^{k+1}-1}{r-1}$.
- **Harmonic Series** $\sum_{i=1}^{n} \frac{1}{i} \approx \ln(n) + \gamma$, where $\gamma = .577$ is called Euler's constant.
- **Arithmetic Geometric Series** $\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$.
- **Fibonacci Numbers** $F_0 = 0$, $F_1 = 1$, for all $k \geq 2$, $F_n = F_{n-1} + F_{n-2}$.

Analysis Of Algorithms

Criteria for selecting algorithms

1. Correctness
2. Amount of work done
3. Amount of space used
4. Simplicity, clarity
5. Optimality

Correctness

Proving correctness is dreadful for large algorithms. A strategy that can be used is: divide the algorithm into smaller pieces, and then clarify what the preconditions and postconditions are and prove correct assuming everything else is correct.

Amount of work done

Rather than counting the total number of instructions executed, we'll focus on a set of key instructions and count how many times they are executed. Use asymptotic notation and pay attention only to the largest growing factor in the formula of the running time.

Two major types of analysis: **worst-case analysis** and **average-case analysis**

- Worst-case analysis: For each $n \geq 1$, let D_n the set of all instances of size n. For each input I, let $t(I)$ be the number of fundamental operations executed on input I. Then, the worst-case running time is given by the function W, defined for all $n \geq 1$, by:

$$W(n) = \max\{t(I) \mid I \in D_n\}$$

- Average-case analysis: For each $n \geq 1$ and $I \in D_n$, let $P(I)$ be the probability that input I is selected when the input size is n. Then, the average-case running time is given by the function A, defined for all $n \geq 1$, by:

$$A(n) = \sum_{I \in D_n} P(I)t(I).$$

Note: According to Dr. Sten Odenwald (http://itss.raytheon.com/cafe/qadir/q1797.html) the number of atoms in the universe is 3×10^{78}. The lg of this is 260.69.... So, a 264-bit address space will be "more than sufficient" to do computation for our universe. 264 bits are only 33 bytes. So, it is OK to assume that the indices require no more than a constant number of bits for representation.

Amount of space used

The amount of space used can be measured similarly. Consideration of this efficiency is often important.

Simplicity, clarity

Sometimes, complicated and long algorithms can be simplified and shortened by the use of recursive calls.

Optimality

For some algorithms, you can argue that they are the "best" in terms of either amount of time used or amount of space used. There are also problems for which you cannot hope to have efficient algorithms.

Asymptotic Growth Rates of Functions

Consider functions from the nonnegative numbers to the nonnegative reals.

For such a function g, we think of the following five function classes consisting of functions whose growth rates are comparable to that of g:

- $O(g)$: $f \in O(g)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.
- $o(g)$: $f \in o(g)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- $\Omega(g)$: $f \in \Omega(g)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$.
- $\omega(g)$: $f \in \omega(g)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.
- $\Theta(f)$: $\Theta(f) = \Omega(g) \cap O(g)$. Alternatively, $f \in \Theta(g)$ if and only if there exists a constant $c > 0$ such that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$.

An Example: Searching in a Sorted Array

Input: An array E of n entries, indexed in the range, first . . . last, and a key K
Output: An integer k such that $E[k] = K$ if such a k exists and -1 otherwise

Let n be the size of the range. n is equal to $\max\{0, \text{last} - \text{first} + 1\}$.

Measure the amount of time used by counting the number of comparisons against K.

Approach 1: Sequential Search

Examine all the entries.

The amount of time used is n because each entry is examined once.

Approach 2: Jumping Search

Examine every jth entry, i.e., the entries at positions first $+ mj - 1$ for $1 \leq m \leq \lfloor n/j \rfloor$.
If K is found the search is over.
As soon as the examined key exceeds K, we know that K should be between the current key and the key examined the last time. So, execute the brute-force search in that region.

The amount of time used is $\Theta((j-1) + n/j)$.
This is minimized when $j = \Theta(\sqrt{n})$. So, set $j = \lceil \sqrt{n} \rceil$. Then the time function is $\Theta(\sqrt{n})$.

Approach 3: Binary Search

Examine the middle point. If that entry is equal to K, the search is over. Otherwise, pick one of the two regions (before or after the middle point) that may contain K (the other can't contain K), and repeat the process.

The number of examinations is at most $\lg(n)$ because the size of the range the next time around is at most a half of the current range. So, the amount of time used is $\Theta(\lg n)$.

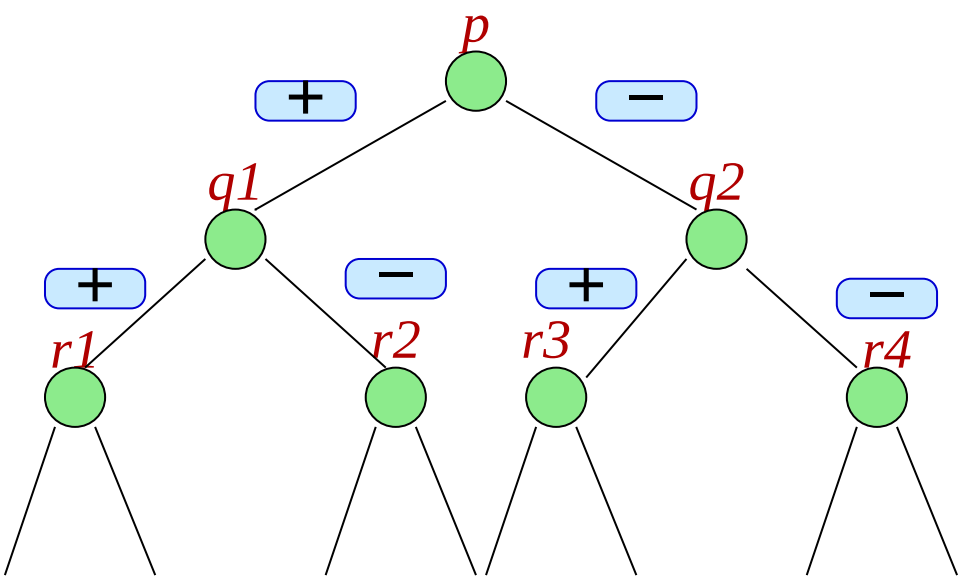
Optimality of Binary Search

We can assume that one examination gives with only three possible types of information:

- the entry is equal to K,
- the entry is greater than K,
- the entry is less than K.

Assume there is an algorithm A that runs in time $T(n) \in o(\log n)$. Let n be such that $2^{T(n)} < n$. We'll consider the behavior of A on inputs of size n not containing the key K that is searched for.

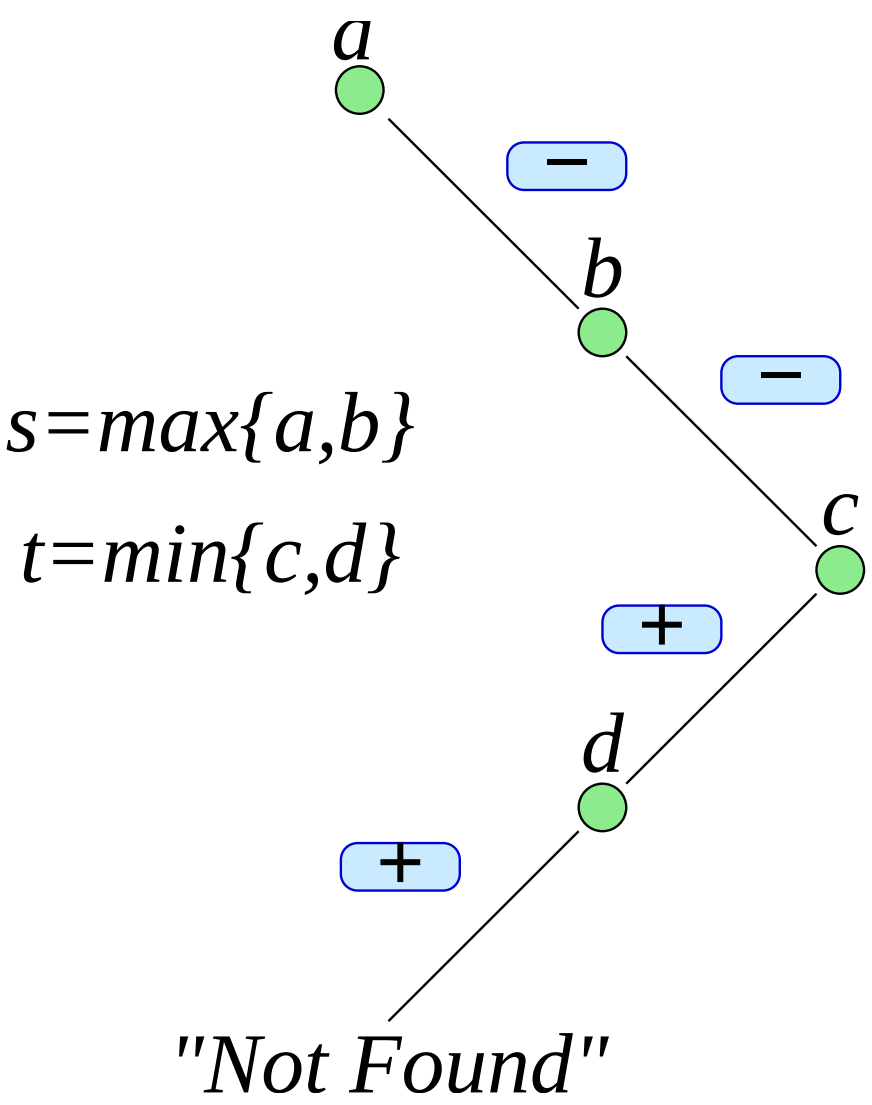
The behavior can be viewed as a binary tree having depth $T(n)$.


p
+
−
q1
q2
+
−
+
−
r1
r2
r3
r4

Let v be an arbitrary leaf and π be the downward path from the root to v. The path represents the sequence of outcomes occurring during a single execution of A.

Let s be the largest examination point along the path for which the key examined is less than K. If there is no such point, let $s = -1$.

Let t be the smallest examination point along the path for which the key examined is greater than K. If there is no such point, let $t = n$.


a
b
c
d
−
−
+
+
s=max{a,b}
t=min{c,d}
"Not Found"

We can assume that $s < t$. Otherwise, the path will never be followed.

Also, we can assume that $s + 1 \geq t$. Otherwise, the range $[s + 1, t - 1]$ is never examined, and thus, we cannot be sure that the key does not exist.

Thus, $s + 1 = t$. This means that at each leaf A discovers that **up to a point the entries are negative and beyond that point the entries are positive.**

The key K corresponds to a unique leaf because of the properties of s and t. Each value between -1 and $n-1$ is possible for s. So, there are $n+1$ possibilities for s. However, the number of leaves in this tree is $2^{T(n)} < n$, so **the algorithm is able to identify not more than n such points.** This means that A is flawed.

Questions

1. As an alternative to binary search, one can think of searching by examining the entries at three places each time, thereby reducing the size of the array by a factor of $1/4$ instead of $1/2$. Is this alternative better or worse?
2. As an alternative to binary search, one can think of searching by examining at $(\lceil \lg n \rceil - 1)$ places in each round thereby reducing the size of the array by a factor of $\lceil \lg n \rceil$. What is the asymptotic order of the number of examinations made?