CS520 Advanced Analysis of Algorithms and Complexity

- Dr. Ben Choi
- Ph.D. in EE (Computer Engineering), The Ohio State University
- System Performance Engineer, Lucent Technologies - Bell Labs Innovations
- Pilot, FAA certified pilot for airplanes and helicopters
  - BenChoi.info

TECH
Computer Science

What is a Computer Algorithm?

- A computer algorithm is
  - **a detailed step-by-step method for**
  - **solving a problem**
  - **by using a computer.**

Problem-Solving (Science and Engineering)

- Analysis
  - **How does it work?**
  - **Breaking a system down to known components**
  - **How the components relate to each other**
  - **Breaking a process down to known functions**
- Synthesis
  - **Building tools and toys!**
  - **What components are needed**
  - **How the components should be put together**
  - **Composing functions to form a process**

Problem Solving Using Computers

- Problem:
- Strategy:
- Algorithm:
  - **Input:**
  - **Output:**
  - **Step:**
- Analysis:
  - **Correctness:**
  - **Time & Space:**
  - **Optimality:**
- Implementation:
- Verification:

Example: Search in an unordered array

- Problem:
  - **Let E be an array containing n entries, E[0], …, E[n-1], in no particular order.**
  - **Find an index of a specified key K, if K is in the array;**
  - **return –1 as the answer if K is not in the array.**
- Strategy:
  - **Compare K to each entry in turn until a match is found or the array is exhausted.**
  - **If K is not in the array, the algorithm returns –1 as its answer.**

Example: Sequential Search, Unordered

- Algorithm (and data structure)
  - **Input: E, n, K, where E is an array with n entries (indexed 0, …, n-1), and K is the item sought. For simplicity, we assume that K and the entries of E are integers, as is n.**
  - **Output: Returns ans, the location of K in E (-1 if K is not found.)**

Algorithm: Step (Specification)

- int seqSearch(int[] E, int n, int K)
- 1. int ans, index;
- 2. ans = -1; // Assume failure.
- 3. for (index = 0; index < n; index++)
- 4. if (K == E[index])
- 5. ans = index; // Success!
- 6. break; // Done!
- 7. return ans;

Analysis of the Algorithm

- How shall we measure the amount of work done by an algorithm?
- Basic Operation:
  - **Comparison of x with an array entry**
- Worst-Case Analysis:
  - **Let W(n) be a function. W(n) is the maximum number of basic operations performed by the algorithm on any input size n.**
  - **For our example, clearly W(n) = n.**
  - **The worst cases occur when K appears only in the last position in the array and when K is not in the array at all.**

More Analysis of the Algorithm

- Average-Behavior Analysis:
  - **Let q be the probability that K is in the array**
  - **A(n) = n(1 – ½ q) + ½ q**
- Optimality:
  - **The Best possible solution?**
  - **Searching an Ordered Array**
  - **Using Binary Search**
  - **W(n) = Ceiling[lg(n+1)]** $= \lceil \lg(n+1) \rceil$
  - **The Binary Search algorithm is optimal.**
- Correctness: (Proving Correctness of Procedures s3.5)

What is CS 520?

- Class Syllabus

Algorithm Language (Specifying the Steps)

- Java as an algorithm language
- Syntax similar to C++
- Some steps within an algorithm may be specified in pseduocode (English phrases)
- Focus on the strategy and techniques of an algorithm, not on detail implementation

Analysis Tool: Mathematics: Set

- A set is a collection of distinct elements.
- The elements are of the same "type", common properties.
- "element e is a member of set S" is denoted as $e \in S$
- Read "e is in S"
- A particular set is defined by listing or describing its elements between a pair of curly braces:
  $S_1 = \{a, b, c\}$, $S_2 = \{x \mid x \text{ is an integer power of } 2\}$
  read "*the set of* all elements x *such that* x is …"
- $S_3 = \{\} = \varnothing$, has not elements, called empty set
- A set has no inherent order.

Subset, Superset; Intersection, Union

- If all elements of one set, S_1
  - are also in another set, S_2,
- Then S_1 is said to be a *subset* of S_2, $S_1 \subseteq S_2$
  - and S_2 is said to be a *superset* of S_1, $S_2 \supseteq S_1$.
- Empty set is a subset of every set, $\varnothing \subseteq S$
- *Intersection*
- $S \cap T = \{x \mid x \in S \text{ and } x \in T\}$
- *Union*
- $S \cup T = \{x \mid x \in S \text{ or } x \in T\}$

Cardinality

- Cardinality
  - **A set, S, is *finite* if there is an integer n such that the elements of S can be placed in a one-to-one correspondence with {1, 2, 3, …, n}**
  - **in this case we write |S| = n**
- How many distinct subsets does a finite set on n elements have? There are 2^n subsets.
- How many distinct subsets of cardinality k does a finite set of n elements have?
  There are C(n, k) = n!/((n-k)!k!), "n choose k" $\binom{n}{k}$

Sequence

- A group of elements in a *specified order* is called a sequence.
- A sequence can have repeated elements.
- Sequences are defined by listing or describing their elements in order, enclosed in parentheses.
- e.g. S1 = (a, b, c), S2 = (b, c, a), S3 = (a, a, b, c)
- A sequence is *finite* if there is an integer *n* such that the elements of the sequence can be placed in a one-to-one correspondence with (1, 2, 3, …, n).
- If all the elements of a finite sequence are distinct, that sequence is said to be a *permutation* of the finite set consisting of the same elements.
- One set of n elements has n! distinct permutations.

Tuples and Cross Product

- A tuple is a finite sequence.
  - **Ordered pair (x, y), triple (x, y, z), quadruple, and quintuple**
  - **A k-tuple is a tuple of k elements.**
- The *cross product* of two sets, say S and T, is
  $S \times T = \{(x, y) \mid x \in S, y \in T\}$
- $|S \times T| = |S|\,|T|$
- It often happens that S and T are the same set, e.g.
  $N \times N$
  where N denotes the set of natural numbers, {0,1,2,…}

Relations and Functions

- A *relation* is some subset of a (possibly iterated) cross product.
- A binary relation is some subset of a cross product, e.g. $R \subseteq S \times T$
- e.g. "less than" relation can be defined as
  $\{(x, y) \mid x \in N, y \in N, x < y\}$
- Important properties of relations; let $R \subseteq S \times S$
  - **reflexive: for all $x \in S$, $(x, x) \in R$.**
  - **symmetric: if $(x, y) \in R$, then $(y, x) \in R$.**
  - **antisymmetric: if $(x, y) \in R$, then $(y, x) \notin R$**
  - **transitive: if $(x,y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$.**
- A relation that is reflexive, symmetric, and transitive is called an *equivalence relation,* partition the underlying set S into equivalence classes $[x] = \{y \in S \mid x R y\}$, $x \in S$
- A *function* is a relation in which no element of S (of S x T) is repeated with the relation. (informal def.)

Analysis Tool: Logic

- Logic is a system for formalizing natural language statements so that we can reason more accurately.
- The simplest statements are called *atomic formulas*.
- More complex statements can be build up through the use of *logical connectives*: $\wedge$ "and", $\vee$ "or", $\neg$ "not", $\Rightarrow$ "implies" $A \Rightarrow B$ "A implies B" "if A then B"
- $A \Rightarrow B$ is logically equivalent to $\neg A \vee B$
- $\neg (A \wedge B)$ is logically equivalent to $\neg A \vee \neg B$
- $\neg (A \vee B)$ is logically equivalent to $\neg A \wedge \neg B$

Quantifiers: all, some

- "for all x" $\forall x\ P(x)$ is true iff $P(x)$ is true for *all* x
  - **universal quantifier (universe of discourse)**
- "there exist x" $\exists x\ P(x)$ is true iff $P(x)$ is true for *some* value of x
  - **existential quantifier**
- $\forall x\ A(x)$ is logically equivalent to $\neg \exists x(\neg A(x))$
- $\exists x\ A(x)$ is logically equivalent to $\neg \forall x(\neg A(x))$
- $\forall x\ (A(x) \Rightarrow B(x))$
  "For all x such that if A(x) holds then B(x) holds"

Prove: by counterexample, Contraposition, Contradiction

- *Counterexample*
  to prove $\forall x\ (A(x) \Rightarrow B(x))$ is false, we show *some* object x for which A(x) is true and B(x) is false.
  - $\neg(\forall x\ (A(x) \Rightarrow B(x))) \Leftrightarrow \exists x\ (A(x) \wedge \neg B(x))$
- *Contraposition*
  to prove $A \Rightarrow B$, we show $(\neg B) \Rightarrow (\neg A)$
- *Contradiction*
  to prove $A \Rightarrow B$, we assume $\neg B$ and then prove B.
  - $A \Rightarrow B \Leftrightarrow (A \wedge \neg B) \Rightarrow B$
  - $A \Rightarrow B \Leftrightarrow (A \wedge \neg B)$ **is false**
  - **Assuming $(A \wedge \neg B)$ is true, and discover a *contradiction* (such as $A \wedge \neg A$), then conclude $(A \wedge \neg B)$ is false, and so $A \Rightarrow B$.**

Prove: by Contradiction, e.g.

- Prove $[B \wedge (B \Rightarrow C)] \Rightarrow C$
  - **by contradiction**
- Proof:
- Assume $\neg C$
- $\neg C \wedge [B \wedge (B \Rightarrow C)]$
- $\Rightarrow \neg C \wedge [B \wedge (\neg B \vee C)]$
- $\Rightarrow \neg C \wedge [(B \wedge \neg B) \vee (B \wedge C)]$
- $\Rightarrow \neg C \wedge [(B \wedge C)]$
- $\Rightarrow \neg C \wedge C \wedge B$
- $\Rightarrow$ False, *Contradiction*
- $\Rightarrow C$

Rules of Inference

- A rule of inference is a *general pattern* that allows us to draw some new conclusion from a set of given statements.
  - **If we know {…} then we can conclude {…}**
- If $\{B \text{ and } (B \Rightarrow C)\}$ then $\{C\}$
  - **modus ponens**
- If $\{A \Rightarrow B \text{ and } B \Rightarrow C\}$ then $\{A \Rightarrow C\}$
  - **syllogism**
- If $\{B \Rightarrow C \text{ and } \neg B \Rightarrow C\}$ then $\{C\}$
  - **rule of cases**

Two-valued Boolean (algebra) logic

- 1. There exists two elements in B, i.e. B={0,1}
  - **there are two binary operations + "or, $\vee$", $\cdot$ "and, $\wedge$"**
- 2. Closure: if $x, y \in B$ and $z = x + y$ then $z \in B$
  - **if $x, y \in B$ and $z = x \cdot y$ then $z \in B$**
- 3. Identity element: for + designated by 0: $x + 0 = x$
  - **for $\cdot$ designated by 1: $x \cdot 1 = x$**
- 4. Commutative: $x + y = y + x$
  - $x \cdot y = y \cdot x$
- 5. Distributive: $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
  - $x + (y \cdot z) = (x + y) \cdot (x + z)$
- 6. Complement: for every element $x \in B$, there exits an element $x' \in B$
  - $x + x' = 1,\ x \cdot x' = 0$

True Table and Tautologically Implies e.g.

- Show $[B \wedge (B \Rightarrow C)] \Rightarrow C$ is a tautology:

B	C	$(B \Rightarrow C)$	$[B \wedge (B \Rightarrow C)]$	$[B \wedge (B \Rightarrow C)] \Rightarrow C$
0	0	1	0	1
0	1	1	0	1
1	0	0	0	1
1	1	1	1	1

- For every assignment for B and C,
  - **the statement is True**

Prove: by Rule of inferences

- Prove $[B \wedge (B \Rightarrow C)] \Rightarrow C$
  - **Proof:**
  - $[B \wedge (B \Rightarrow C)] \Rightarrow C$
  - $\Rightarrow \neg[B \wedge (B \Rightarrow C)] \vee C$
  - $\Rightarrow \neg[B \wedge (\neg B \vee C)] \vee C$
  - $\Rightarrow \neg[(B \wedge \neg B) \vee (B \wedge C)] \vee C$
  - $\Rightarrow \neg [(B \wedge C)] \vee C$
  - $\Rightarrow \neg B \vee \neg C \vee C$
  - $\Rightarrow$ **True (*tautology*)**
- Direct Proof:
  - $[B \wedge (B \Rightarrow C)] \Rightarrow [B \wedge C] \Rightarrow C$

Analysis Tool: Probability

- Elementary events (outcomes)
  - **Suppose that in a given situation an event, or experiment, may have any one, and only one, of k outcomes, $s_1, s_2, \ldots, s_k$. (mutually exclusive)**
- Universe
  The set of all elementary events is called the *universe* and is denoted $U = \{s_1, s_2, \ldots, s_k\}$.
- Probability of s_i
- associate a real number $Pr(s_i)$, such that
- $0 \le Pr(s_i) \le 1$ for $1 \le i \le k$;
- $Pr(s_1) + Pr(s_2) + \ldots + Pr(s_k) = 1$

Event

- Let $S \subseteq U$. Then S is called an *event*, and
- $Pr(S) = \sum_{si \in S} Pr(s_i)$
- Sure event $U = \{s_1, s_2, \ldots, s_k\}$, $Pr(U) = 1$
- Impossible event, $\varnothing$,$Pr(\varnothing) = 0$
- Complement event "not S" U – S,
  Pr(not S) = 1 – Pr(S)

Conditional Probability

- The conditional probability of an event S *given* an event T is defined as
- $Pr(S \mid T) = Pr(S \text{ and } T) / Pr(T)$
  $= \sum_{si \in S \cap T} Pr(s_i) / \sum_{sj \in T} Pr(s_j)$
- *Independent*
- Given two events S and T, if
- $Pr(S \text{ and } T) = Pr(S)Pr(T)$
- then S and T are stochastically independent, or simply independent.

Random variable and their Expected value

- A *random variable* is a real valued variable that depends on which elementary event has occurred
  - **it is a function defined for elementary events.**
  - **e.g. f(e) = the number of inversions in the permutation of {A, B, C}; assume all input permutations are equally likely.**
- Expectation
  - **Let f(e) be a random variable defined on a set of elementary events $e \in U$. The expectation of f, denoted as E(f), is defined as**
- $E(f) = \sum_{e \in U} f(e)Pr(e)$
  - **This is often called the average values of f.**
  - **Expectations are often easier to manipulate then the random variables themselves.**

Conditional expectation and Laws of expectations

- The *conditional expectation* of f given an event S, denoted as $E(f \mid S)$, is defined as
- $E(f \mid S) = \sum_{e \in S} f(e)Pr(e \mid S)$
- *Law of expectations*
- For random variables f(e) and g(e) defined on a set of elementary events $e \in U$, and any event S:
- $E(f + g) = E(f) + E(g)$
- $E(f) = Pr(S)E(f \mid S) + Pr(\text{not } S)\, E(f \mid \text{not } S)$

Analysis Tool: Algebra

- *Manipulating Inequalities*
- Transitivity: If ((A ≤ B) and (B ≤ C) Then (A ≤ C)
- Addition: If ((A ≤ B) and (C ≤ D) Then (A+C ≤ B+D)
- Positive Scaling:
  If ((A ≤ B) and (α > 0) Then (α A ≤ α B)
- *Floor and Ceiling Functions*
- Floor[x] is the largest integer less than or equal to x.
  $\lfloor x \rfloor$
- Ceiling[x] is the smallest integer greater than or equal to x. $\lceil x \rceil$

Logarithms

- **For b>1 and x>0,**
  $\log_b x$ (read "log to the base b of x")
  is that real number L such that $b^L = x$
- **$\log_b x$ is the power to which b must be raised to get x.**
- Log properties: def: $\lg x = \log_2 x$; $\ln x = \log_e x$
  - Let x and y be arbitrary positive real numbers, let a, b any real number, and let b>1 and c>1 be real numbers.
- **$\log_b$ is a strictly increasing function,**
  if x > y then $\log_b x > \log_b y$
- **$\log_b$ is a one-to-one function,**
  if $\log_b x = \log_b y$ then x = y
- **$\log_b 1 = 0$; $\log_b b = 1$; $\log_b x^a = a \log_b x$**
- **$\log_b(xy) = \log_b x + \log_b y$**
- **$x^{\log y} = y^{\log x}$**
- **change base: $\log_c x = (\log_b x)/(\log_b c)$**

Series

- A *series* is the sum of a sequence.
- Arithmetic series $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
  - **The sum of consecutive integers**
- Polynomial Series $\sum_{i=1}^{n} i^2 = \frac{2n^3+3n^2+n}{6} \approx \frac{n^3}{3}$
  - **The sum of squares**
  - **The general case is** $\sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{k+1}$
- Power of 2 $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$
- Arithmetic-
- Geometric Series $\sum_{i=1}^{k} i2^i = (k-1)2^{k+1} + 2$

Summations Using Integration

- **A function f(x) is said to be monotonic, or *nondecreasing*, if x ≤ y always implies that f(x) ≤ f(y).**
- **A function f(x) is antimonotonic, or *nonincreasing*, if –f(x) is monotonic.**
- If f(x) is nondecreasing then

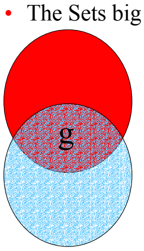
$$\int_{a-1}^{b} f(x)dx \le \sum_{i=a}^{b} f(i) \le \int_{a}^{b+1} f(x)dx$$

- If f(x) is nonincreasing then

$$\int_{a}^{b+1} f(x)dx \le \sum_{i=a}^{b} f(i) \le \int_{a-1}^{b} f(x)dx$$

Classifying functions by their Asymptotic Growth Rates

- asymptotic growth rate, asymptotic order, or order of functions
  - **Comparing and classifying functions that ignores *constant factors* and *small inputs*.**
- The Sets big oh O(g), big theta Θ(g), big omega Ω(g)



Ω(g): functions that grow **at least as fast** as g

Θ(g): functions that grow **at the same rate** as g

O(g): functions that grow **no faster** than g

The Sets O(g), Θ(g), Ω(g)

- **Let *g* and *f* be a functions from the nonnegative integers into the positive real numbers**
- **For some real constant c > 0 and some nonnegative integer constant n_0**
- O(g) is the set of functions f, such that
- $f(n) \le c\, g(n)$ for all $n \ge n_0$
- Ω(g) is the set of functions f, such that
- $f(n) \ge c\, g(n)$ for all $n \ge n_0$
- Θ(g) = O(g) ∩ Ω(g)
  - **asymptotic order of g**
  - **f ∈Θ(g) read as "f is asymptotic order g" or "f is order g"**

Comparing asymptotic growth rates

- Comparing f(n) and g(n) as n approaches infinity,
- IF

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

- $< \infty$, including the case in which the limit is 0 then $f \in O(g)$
- > 0, including the case in which the limit is ∞ then $f \in \Omega(g)$
- $= c$ and $0 < c < \infty$ then $f \in \Theta(g)$
- $= 0$ then $f \in o(g)$ //read as "little oh of g"
- $= \infty$ then $f \in \omega(g)$ //read as "little omega of g"

Properties of O(g), Θ(g), Ω(g)

- Transitive: If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$. O is transitive. Also Ω, Θ, o, ω are transitive.
- Reflexive: $f \in \Theta(f)$
- Symmetric: If $f \in \Theta(g)$, then $g \in \Theta(f)$
- Θ defines an equivalence relation on the functions.
  - **Each set Θ(f) is an equivalence class (complexity class).**
- $f \in O(g) \Leftrightarrow g \in \Omega(f)$
- $O(f + g) = O(\max(f, g))$
  similar equations hold for Ω and Θ

Classification of functions, e.g.

- O(1) denotes the set of functions bounded by a *constant* (for large n)
- $f \in \Theta(n)$, f is *linear*
- $f \in \Theta(n^2)$, f is *quadratic*; $f \in \Theta(n^3)$, f is *cubic*
- $\lg n \in o(n^\alpha)$ for any $\alpha > 0$, including factional powers
- $n^k \in o(c^n)$ for any $k > 0$ and any $c > 1$
  - **powers of n grow more slowly than any exponential function c^n**

$$\sum_{i=1}^{n} i^d \in \Theta(n^{d+1}) \qquad \sum_{i=1}^{n} \log(i) \in \Theta(n\log(n))$$

$$\sum_{i=a}^{b} r^i \in \Theta(r^b) \text{ for } r > 0,\ r \neq 1, \text{ b may be some function of n}$$

Analyzing Algorithms and Problems

- We analyze algorithms with the intention of *improving* them, if possible, and for *choosing* among several available for a problem.
- Correctness
- Amount of work done, and space used
- Optimality, Simplicity

Correctness can be proved!

- An algorithm consists of sequences of steps (operations, instructions, statements) for *transforming* inputs (preconditions) to outputs (postconditions)
- Proving
  - **if the preconditions are satisfied,**
  - **then the postconditions will be true,**
  - **when the algorithm terminates.**

Amount of work done

- **We want a measure of work that tells us something about the *efficiency* of the method used by the algorithm**
- **independent of computer, programming language, programmer, and other implementation details.**
- **Usually depending on the *size of the input***
- Counting passes through loops
- Basic Operation
  - **Identify a particular operation fundamental to the problem**
  - **the total number of operations performed is roughly proportional to the number of basic operations**
- Identifying the properties of the inputs that affect the behavior of the algorithm

Worst-case complexity

- Let D_n be *the set of inputs of size n* for the problem under consideration, and let I be an element of D_n.
- Let t(I) be *the number of basic operations* performed by the algorithm on input I.
- We define the function W by

- $W(n) = \max\{t(I) \mid I \in D_n\}$
  - called the worst-case complexity of the algorithm.
  - W(n) is the maximum number of basic operations performed by the algorithm on any input of size n.
- The input, I, for which an algorithm behaves worst depends on the particular algorithm.

Average Complexity

- Let Pr(I) be the *probability* that input I occurs.
- Then the average behavior of the algorithm is defined as

- $A(n) = \Sigma_{I \in Dn} Pr(I)\, t(I)$.
  - We determine t(I) by analyzing the algorithm,
  - but Pr(I) cannot be computed analytically.
- $A(n) = Pr(succ)A_{succ}(n) + Pr(fail)A_{fail}(n)$
- An element I in D_n may be thought as a set or equivalence class that affect the behavior of the algorithm. (see following e.g. n+1 cases)

e.g. Search in an unordered array

- int seqSearch(int[] E, int n, int K)
- 1. int ans, index;
- 2. ans = -1; // Assume failure.
- 3. for (index = 0; index < n; index++)
- 4. if (K == E[index])
- 5. ans = index; // Success!
- 6. break; // Done!
- 7. return ans;

Average-Behavior Analysis e.g.

- $A(n) = Pr(succ)A_{succ}(n) + Pr(fail)A_{fail}(n)$
- There are total of n+1 cases of I in D_n
  - Let K is in the array as "succ" cases that have n cases.
  - Assuming K is equally likely found in any of the n location, i.e. $Pr(I_i \mid succ) = 1/n$
  - for $0 <= i < n$, $t(I_i) = i + 1$
  - $A_{succ}(n) = \Sigma_{i=0}^{n-1} Pr(I_i \mid succ)\, t(I_i)$
  - $= \Sigma_{i=0}^{n-1} (1/n)\,(i+1) = (1/n)[n(n+1)/2] = (n+1)/2$
  - Let K is not in the array as the "fail" case that has 1 cases, $Pr(I \mid fail) = 1$
  - Then $A_{fail}(n) = Pr(I \mid fail)\, t(I) = 1\ n$
- Let q be the probability for the succ cases
  - $q\,[(n+1)/2] + (1-q)\,n$

Space Usage

- If memory cells used by the algorithms depends on the particular input,
  - then worst-case and average-case analysis can be done.
- Time and Space Tradeoff.

Optimality "the best possible"

- Each problem has inherent complexity
  - There is some *minimum* amount of work required to solve it.
- To analyze the complexity of a problem,
  - we choose a class of algorithms, based on which
  - prove theorems that establish a *lower bound* on the number of operations needed to solve the problem.
- Lower bound (for the worst case)

Show whether an algorithm is optimal?

- Analyze the algorithm, call it A, and found the Worst-case complexity $W_A(n)$, for input of size n.
- Prove a theorem starting that,
  - **for any algorithm in the same class of A**
  - **for any input of size n, there is some input for which the algorithm must perform**
  - **at least $W_{[A]}(n)$ (lower bound in the worst-case)**
- If $W_A(n) == W_{[A]}(n)$
  - **then the algorithm A is optimal**
  - **else there may be a better algorithm**
  - **OR there may be a better lower bound.**

Optimality e.g.

- Problem
  - **Fining the largest entry in an (unsorted) array of n numbers**
- Algorithm A
  - **int findMax(int[] E, int n)**
  - **1. int max;**
  - **2. max = E[0]; // Assume the first is max.**
  - **3. for (index = 1; index < n; index++)**
  - **4.   if (max < E[index])**
  - **5.   max = E[index];**
  - **6. return max;**

Analyze the algorithm, find $W_A(n)$

- Basic Operation
  - **Comparison of an array entry with another array entry or a stored variable.**
- Worst-Case Analysis
  - **For any input of size n, there are exactly n-1 basic operations**
  - **$W_A(n) = n-1$**

For the class of algorithm [A], find $W_{[A]}(n)$

- Class of Algorithms
  - **Algorithms that can compare and copy the numbers, but do no other operations on them.**
- Finding (or proving) $W_{[A]}(n)$
  - **Assuming the entries in the array are all distinct**
    - (permissible for finding lower bound on the worst-case)
  - **In an array with n distinct entries, n – 1 entries are not the maximum.**
  - **To conclude that an entry is not the maximum, it must be smaller than at least one other entry. And, one comparison (basic operation) is needed for that.**
  - **So at least n-1 basic operations must be done.**
  - **$W_{[A]}(n) = n - 1$**
- Since $W_A(n) == W_{[A]}(n)$, algorithm A is optimal.

Simplicity

- Simplicity in an algorithm is a virtue.

Designing Algorithms

- Problem solving using Computer
- Algorithm Design Techniques
  - **divide-and-conquer**
  - **greedy methods**
  - **depth-first search (for graphs)**
  - **dynamic programming**

Problem and Strategy A

- Problem: array search
  - **Given an array E containing n and given a value K, find an index for which K = E[index] or, if K is not in the array, return –1 as the answer.**
- Strategy A
  - **Input data and Data structure: unsorted array**
  - **sequential search**
- Algorithm A
  - **int seqSearch(int[] E, int n, int k)**
- Analysis A
  - **W(n) = n**
  - **A(n) = q [(n+1)/2] + (1-q) n**

Better Algorithm and/or Better Input Data

- Optimality A
  - **for searching an unsorted array**
  - **$W_{[A]}(n) = n$**
  - **Algorithm A is optimal.**
- Strategy B
  - **Input data and Data structure: array sorted in nondecreasing order**
  - **sequential search**
- Algorithm B.
  - **int seqSearch(int[] E, int n, int k)**
- Analysis B
  - **W(n) = n**
  - **A(n) = q [(n+1)/2] + (1-q) n**

Better Algorithm

- Optimality B
  - **It makes no use of the fact that the entries are ordered**
  - **Can we modify the algorithm so that it uses the added information and does less work?**
- Strategy C
  - **Input data and Data structure: array sorted in nondecreasing order**
  - **sequential search:**
    as soon as an entry larger than K is encountered, the algorithm can terminate with the answer –1.

Algorithm C: modified sequential search

```
int seqSearchMod(int[] E, int n, int K)
1. int ans, index;
2. ans = -1; // Assume failure.
3. for (index = 0; index < n; index++)
4.     if (K > E[index])
5.         continue;
6.     if (K < E[index])
7.         break; // Done!
8.     // K == E[index]
9.     ans = index; // Find it
10.    break;
11. return ans;
```

Analysis C

- $W(n) = n + 1 \approx n$
- Average-Behavior
  - **n cases for success:**
  - **$A_{succ}(n) = \sum_{i=0}^{n-1} Pr(I_i \mid succ)\, t(I_i)$**
  - **$= \sum_{i=0}^{n-1} (1/n)\,(i+2) = (3 + n)/2$**
  - **n+1 cases or (gaps) for fail: <E[0]<>E[1]…E[n-1]>**
- $A_{fail}(n) = Pr(I_i \mid fail)\, t(I_i) =$
- $\quad \sum_{i=0}^{n-1} (1/(n+1))\,(i+2) + n/(n+1)$
- $A(n) = q\,(3+n)/2 + (1-q)\,(\,n/(n+1) + (3+n)/2\,)$
- $\quad \approx n/2$

Let's Try Again! Let's divide-and-conquer!

- Strategy D
  - **compare K first to the entry in the middle of the array**
  - **-- eliminates half of the entry with one comparison**
  - **apply the same strategy recursively**
- Algorithm D: Binary Search
  - **Input: E, first, last, and K, all integers, where E is an ordered array in the range first, …, last, and K is the key sought.**
  - **Output: index such that E[index] = K if K is in E within the range first, …, last, and index = -1 if K is not in this range of E**

Binary Search

```
int binarySearch(int[] E, int first, int last, int K)
1.  if (last < first)
2.      index = -1;
3.  else
4.      int mid = (first + last)/2
5.      if (K == E[mid])
6.        index = mid;
7.      else if (K < E[mid])
8.        index = binarySearch(E, first, mid-1, K)
9.      else
10.       index = binarySearch(E, mid+1, last, K);
11. return index
```

Worst-Case Analysis of Binary Search

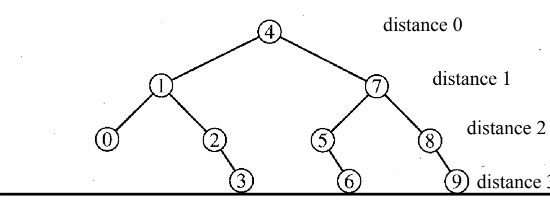
- **Let the problem size be n = last – first + 1; n>0**
- Basic operation is a comparison of K to an array entry
  - **Assume one comparison is done with the three-way branch**
  - **First comparison, assume K != E[mid], divides the array into two sections, each section has at most Floor[n/2] entries.**
  - **estimate that the size of the range is divided by 2 with each recursive call**
  - **How many times can we divide n by 2 without getting a result lest than 1 (i.e. $n/(2^d) >= 1$) ?**
  - **d <= lg(n), therefore we do Floor[lg(n)] comparison following recursive calls, and one before that.**
  - **W(n) = Floor[lg(n)] + 1 = Ceiling[lg(n + 1)] $\in \Theta(\log n)$**

Average-Behavior Analysis of Binary Search

- **There are n+1 cases, n for success and 1 for fail**
- Similar to worst-case analysis, Let $n = 2^d - 1$
  $A_{fail} = \lg(n+1)$
- Assuming $Pr(I_i \mid succ) = 1/n$ for $1 <= i <= n$
  - **divide the n entry into groups, S_t for $1 <= t <= d$, such that S_t requires t comparisons (capital S for group, small s for cardinality of S)**
  - **It is easy to see (?) that (members contained in the group)**
  - **$s_1 = 1 = 2^0$, $s_2 = 2 = 2^1$, $s_3 = 4 = 2^2$, and in general, $s_t = 2^{t-1}$**
  - **The probability that the algorithm does t comparisons is s_t/n**
  - **$A_{succ}(n) = \Sigma_{t=1}^{d} (s_t/n)\, t = ((d-1)2^d + 1)/n$**
  - **d = lg(n+1)**
  - **$A_{succ}(n) = \lg(n+1) - 1 + \lg(n+1)/n$**
- $A(n) \approx \lg(n+1) - q$, where q is probability of successful search

Optimality of Binary Search

- So far we improve from θ(n) algorithm to θ(log n)
  - **Can more improvements be possible?**
- Class of algorithm: comparison as the basic operation
- Analysis by using decision tree, that
  - **for a given input size n is a binary tree whose nodes are labeled with numbers between 0 and n-1 as e.g.**



Decision tree for analysis

- The number of comparisons performed in the worst case is the number of nodes on a longest path from the root to a leaf; call this number p.
- Suppose the decision tree has N nodes
- $N <= 1 + 2 + 4 + \ldots + 2^{p-1}$
- $N <= 2^p - 1$
- $2^p >= (N + 1)$
- Claim N >= n if an algorithm A works correctly in all cases
  - **there is some node in the decision tree labeled i for each i from 0 through n - 1**

Prove by contradiction that N >= n

- Suppose there is no node labeled i for some i in the range from 0 through n-1
  - **Make up two input arrays E1 and E2 such that**
  - **E1[i] = K but E2[i] = K' > K**
  - **For all j < i, make E1[j] = E2[j] using some key values less than K**
  - **For all j > i, make E1[j] = E2[j] using some key values greater than K' in sorted order**
  - **Since no node in the decision tree is labeled i, the algorithm A never compares K to E1[i] or E2[i], but it gives same output for both**
  - **Such algorithm A gives wrong output for at least one of the array and it is not a correct algorithm**
- Conclude that the decision has at least n nodes

Optimality result

- 2^p >= (N+1) >= (n+1)
- p >= lg(n+1)

- Theorem: Any algorithm to find K in an array of n entries (by comparing K to array entries) must do at least Ceiling[lg(n+1)] comparisons for some input.

- Corollary: Since Algorithm D does Ceiling[lg(n+1)] comparisons in the worst case, it is optimal.