Chapter 6 – Asymptotic Analysis – Objectives

- An introduction to the math for determining the theoretical efficiency of an algorithm or computation.
- The commonly-used different orders of magnitude for computations or algorithms.
- Basic principles, that can be intuitively applied without further rigorous proof, to determine the efficiency of an algorithm.
- Distinguishing the leading terms from the irrelevant terms.
- The matching of order-of-magnitude efficiency analysis to the loops in real program code to aid the programmer in deverloping efficient code.
- A demonstration that the binary divide-and-conquer strategy is an optimal strategy for computation and exhibits $O(\lg N)$ complexity.

Efficiency and Complexity

- Computers are fast, but that means we just try to solve bigger problems.
- We need to know about *analysis of algorithms*.
- For example: Binary search is "better" than linear search, but how much better and how do we quantify "better" in a rigorous way?
- We have to look at *worst case* behavior of an algorithm.
- Sometimes we are able to look at *average case* behavior of an algorithm.

What Is a Logarithm?

- *natural logarithm* $\ln x = \log_e x$.
- *common logarithm* $\log_{10} x = \ln x / \ln 10$.
- *binary logarithm* $\lg x = \log_2 x = \ln x / \ln 2$.
- We will show that these are all the same for our purposes here.

Comparing Algorithms

- Binary search is about $\lg N$ comparisons on a list of N items.
- Linear search is about N comparisons on a list of N items.
- The quotient, by L'Hôpital's Rule, has limit

$$\lim_{N\to\infty} \frac{\lg N}{N} = \lim_{N\to\infty} \frac{\frac{1}{N}}{1} = \lim_{N\to\infty} \frac{1}{N} = 0.$$

- The *relative* cost of binary compared to linear goes to zero as the list size goes to infinity.
- And we don't care if, for example, linear takes cost 10^{-100} for each "comparison" and binary takes lots of machine instructions and thus cost 10^{100} for each "comparison," because

$$\lim_{N\to\infty} \frac{10^{100}\lg N}{10^{-100}N} = \lim_{N\to\infty} \frac{10^{200}\frac{1}{N}}{1} = \lim_{N\to\infty} \frac{10^{200}}{N} = 0.$$

Asymptotic Notation

Definition 6.1. *We write*

$$f(x) = O(g(x))$$

if there exist constants c and C such that $x > c \implies |f(x)| \leq C \cdot g(x)$.

- We say that "$f(x)$ is *big Oh* of $g(x)$."
- We showed above for binary versus linear search that $\lg N = O(N)$.

We Can Usually Be a Little Bit Sloppy

- Mathematicians deal with all possible functions.
- We deal with functions that measure work, and thus our functions are almost always positive.
- So we can usually be a little bit sloppy and not write

$$x > c \implies |f(x)| \leq C \cdot g(x)$$

but write instead

$$x > c \implies f(x) \leq C \cdot g(x)$$

.

We Can Usually Be a Little Bit Sloppy (2)

- We do calculus on continuous variables x using L'Hôpital's Rule.
- Functions that measure work usually count things in integer numbers n.
- It is not hard to show that we can ignore the difference between the two so we will not be too careful about this.

Basic Orders of Magnitude and Intuition

- We will show later that each of these is big Oh of the one below it.
- $O(\lg N)$ running time: This is the cost of binary search.
- $O(N)$ running time: This is the cost of a single loop running 1 to N, provided that the body of the loop does a fixed amount of work for each iteration.

```
sum = 0;
for(int i = 0; i < N; ++i)
{
  sum += myList.get(i);
}
```

Basic Orders of Magnitude and Intuition (2)

- $O(N^2)$ running time: This is the cost of a nested double `for` loop that runs from 1 to N in both loops.

  Also the cost of a doubly-nested loop that doesn't quite run all the way to N, such as in a bubblesort.

```
for(int i = 0; i < N-1; ++i)
{
  for(int j = i; j < N; ++j)
  {
    test and maybe swap the i-th and the j-th entries
  }
}
```

Basic Orders of Magnitude and Intuition (3)

- $O(N^3)$ running time: This is the cost of a nested triple `for` loop that runs from 1 to N in both loops.

  Also the cost of a doubly-nested loop that doesn't quite run all the way to N, such as in a bubblesort.

  This often happens in dealing with chemistry or physics in three dimensions in the real world.
- $O(2^N)$ running time: *exponential time* running time.
- $O(N^N)$ running time: (even worse) *exponential time* running time.

Some Numbers

N	$\lg N$	N	N^2	N^3	2^N
16	4	16	256	4096	65536
32	5	32	1024	32768	4294967296
64	6	64	4096	262144	1.8×10^{19}
128	7	128	16384	2097152	3.4×10^{38}
256	8	256	65536	16777216	1.2×10^{77}

Basic Intutive Rules – The Constants Don't Matter

Theorem 6.2. *For any fixed constant K, any function $KF(N)$ is big Oh of $F(N)$.*

Proof. Let $f(x) = KF(N)$, and choose $c = 1$ and $C = K$. This provides us with a function $g(x) = CF(N)$ for which for any N whatsoever

$$f(x) = KF(N) \leq CF(N) = g(x)$$

By definition, this means $f(x) = O(g(x)$. □

The constants come from the "the number of machine instructions" needed to do one X. We are counting the number of X operations, and that count is going off to infinity, so whether an X costs 1 or 50 instructions doesn't matter. The 1 or 50 stays the same at 1 or 50, and it's the part going off to infinity that matters.

Basic Intutive Rules – Constant Startup Costs Don't Matter

• Starting a computation is a fixed constant amount of one-time work.

• Ending a computation is a fixed constant amount of one-time work.

• Total cost is

fixed constant startup + loop cost + plus fixed constant shutdown

Theorem 6.3. *If* $f(x) = O(g(x))$*, and if* $\lim_{x\to\infty} f(x) = \infty$*, then* $f(x) + K = O(g(x))$ *for any constant* K*.*

Proof. By definition, $f(x) = O(g(x))$ means that there are constants c_1 and C_1 such that for all $x > c_1$, we have $|f(x)| \leq C_1 g(x)$. If $\lim_{x\to\infty} f(x) = \infty$, then there is clearly a point x_1 at which $K < f(x_1) < C_1 g(x_1)$, and we can use $2C_1$ as our constant instead of C_1. □

• Notice that we need the condition that $f(x)$ go off to infinity, but since we are looking at functions that measure work, this is always going to be true.

Basic Intutive Rules – Big Oh Is Transitive

• Basically, big Oh works like $\leq$. If algorithm A is big Oh of algorithm B, and algorithm B is big Oh of algorithm C, then algorithm A is also big Oh of algorithm C.

Theorem 6.4. *If* $f(x) = O(g(x))$ *and* $g(x) = O(h(x))$*, then* $f(x) = O(h(x))$.

Proof. By definition, $f(x) = O(g(x))$ means that there are constants c_1 and C_1 such that for all $x > c_1$, we have $|f(x)| \leq C_1 g(x)$. Similarly, from the second assumption we have constants c_2 and C_2 such that for all $x > c_2$ we have $|g(x)| \leq C_2 h(x)$. For all $x > \max(c_1, c_2)$, then we have both assumptions true and thus

$$|f(x)| \leq C_1 g(x) \leq C_1 C_2 h(x)$$

To conclude that $f(x) = O(h(x))$, then, we can choose $c = \max(c_1, c_2)$ and $C = C_1 C_2$. □

Basic Intutive Rules – All Logarithms Are the Same

Theorem 6.5. *For any fixed base a, the function $\log_a N$ is $O(\lg N)$.*

Proof. Let $f(x) = \log_a N = \lg N / \lg a$ by the usual rules of logarithms. Then choose $c = 1$ and $C = 1/\lg a$, and we have a function $g(x) = C \lg N$ for which $f(x) = \log_a N < C \lg N = g(x)$. This is true for all $x > 0$, so we may choose $c = 1$ and conclude that $f(x) = O(g(x)$. □

• This lets us do calculus but then apply without any further proof to binary logs that come from divide and conquer algorithms.

Basic Intutive Rules – All Polynomials of the Same Degree Are the Same

Theorem 6.6. *Let* $f(x) = a_k x^k + a_{k-1}x^{k-1} + ... + a_1 x + a_0$ *be a polynomial of degree* k*. Then* $f(x)$ *is* $O(x^k)$.

Proof. Here, at least to prove this principle, we do need to worry about absolute values. Let $A = \max\{|a_i|\}$ be the maximum in absolute value of the coefficients of $f(x)$. Let $g(x) = k \cdot A \cdot x^k$. By our choice of A we have for all $x \geq 1$ that

$$\begin{aligned} f(x) &= a_k x^k + a_{k-1}x^{k-1} + ... + a_1 x + a_0 \\ &\leq |a_k x^k + a_{k-1}x^{k-1} + ... + a_1 x + a_0| \end{aligned}$$

since any value z is less than or equal to its absolute value $|z|$.

Basic Intutive Rules – All Polynomials of the Same Degree Are the Same (2)

We can apply the triangle inequality[1] to distribute the absolute values:

$$\begin{aligned} f(x) &= a_k x^k + a_{k-1} x^{k-1} + ... + a_1 x + a_0 \\ &\leq |a_k x^k + a_{k-1} x^{k-1} + ... + a_1 x + a_0| \\ &\leq |a_k| x^k + |a_{k-1}| x^{k-1} + ... + |a_1| x + |a_0| \end{aligned}$$

and now we get to be rather sloppy. We are only interested in positive values of x (which is n, after all), and for any positive x we have $x^{m-1} \leq x^m$. So we can raise all the exponents to the highest power exponent to get

$$\begin{aligned} f(x) &= a_k x^k + a_{k-1} x^{k-1} + ... + a_1 x + a_0 \\ &\leq |a_k x^k + a_{k-1} x^{k-1} + ... + a_1 x + a_0| \\ &\leq |a_k| x^k + |a_{k-1}| x^{k-1} + ... + |a_1| x + |a_0| \\ &\leq |a_k| x^k + |a_{k-1}| x^k + ... + |a_1| x^k + |a_0| x^k \end{aligned}$$

[1] $|A + B| \leq |A| + |B|$

Basic Intutive Rules – All Polynomials of the Same Degree Are the Same (3)

Now we can be sloppy yet again. Each of the coefficients is smaller in absolute value than A, so we replace all the coefficients with A:

$$\begin{aligned}
f(x) &= a_k x^k + a_{k-1}x^{k-1} + ... + a_1 x + a_0 \\
&\leq |a_k x^k + a_{k-1}x^{k-1} + ... + a_1 x + a_0| \\
&\leq |a_k| x^k + |a_{k-1}| x^{k-1} + ... + |a_1| x + |a_0| \\
&\leq |a_k| x^k + |a_{k-1}| x^k + ... + |a_1| x^k + |a_0| x^k \\
&\leq Ax^k + Ax^k + ... + Ax^k + Ax^k \\
&= kAx^k \\
&= g(x)
\end{aligned}$$

By definition, $f(x) = O(g(x))$. □

- This lets us ignore everything except the term of highest exponent.

Basic Intutive Rules – All Polynomials of the Same Degree Are the Same (4)

- We can prove this in a different way.

  We can clearly write

$$\begin{aligned} f(x) &= a_k x^k + a_{k-1} x^{k-1} + ... + a_1 x + a_0 \\ &\leq |a_k x^k + a_{k-1} x^{k-1} + ... + a_1 x + a_0| \\ &\leq |a_k| x^k + |a_{k-1}| x^{k-1} + ... + |a_1| x + |a_0| \\ &\leq A x^k + A x^{k-1} + ... + A x + A \end{aligned}$$

  But instead of being heavy handed as before, let's roll up the terms from the right one by one, repeatedly using the fact that for positive x we have $x^{m-1} \leq x^m$:

Basic Intutive Rules – All Polynomials of the Same Degree Are the Same (5)

$$
\begin{aligned}
f(x) &= a_k x^k + a_{k-1} x^{k-1} + ... + a_1 x + a_0 \\
&\leq |a_k x^k + a_{k-1} x^{k-1} + ... + a_1 x + a_0| \\
&\leq |a_k| x^k + |a_{k-1}| x^{k-1} + ... + |a_1| x + |a_0| \\
&\leq A x^k + A x^{k-1} + ... + A x + A \\
&\leq A x^k + A x^{k-1} + ... + A x^2 + 2Ax \\
&\leq A x^k + A x^{k-1} + ... + A x^3 + +3Ax^2 \\
&... \\
&\leq A x^k + (k-1) A x^{k-1} \\
&\leq k A x^k
\end{aligned}
$$

The result is the same.

Basic Intutive Rules – All Reasonable Cost-of-Work Functions Go Off to Infinity

• If you are doing something that requires determining whether or not a property holds for a data set of N items, then you cannot determine whether or not the property holds without looking at all N items.

• Each look costs you at least 1, so the total cost of looking at N items has to be at least N, which goes to infinity with N.

Little oh

- Big Oh is a "less-than-or-equal-to."
- Little oh works like "genuinely less than."
-

Definition 6.7. *We write*

$$f(x) = o(g(x))$$

if $\lim_{x\to\infty} \frac{|f(x)|}{|g(x)|} = 0$.

We say that $f(x)$ is "little oh" of $g(x)$.

- Intuitively, what this means is that the relative size of $f(x)$ becomes vanishingly small compared to $g(x)$ as x gets large.
- This lets us say that one algorithm is "genuinely better" than another.

Little oh Implies Big Oh

Theorem 6.8. *If we have $f(x) = o(g(x))$ and $g(x)$ is a positive function everywhere, then we have $f(x) = O(g(x))$.*

Proof. If $\lim_{x\to\infty} \frac{|f(x)|}{|g(x)|} = 0$, then by the definition of limit we have that for any positive constant C, there exists a constant c such that for $x > c$ we have $|f(x)| \leq C \cdot g(x)$. This is much stronger than what we need to show big Oh, which is only that we have such a bound for *some* C. □

Theorem 6.9. *For any positive integer k and for any $\varepsilon > 0$, we have*

$$x^k = o(x^{k+\varepsilon}).$$

Proof. Clearly,

$$\lim_{x\to\infty} \frac{x^k}{x^{k+\varepsilon}} = \lim_{x\to\infty} \frac{1}{x^{\varepsilon}} = 0.$$

□

The Hierarchy of Orders of Magnitude (2)

We can also prove that $\lg N = o(N)$ by using L'Hôpital's rule. We have

$$\lim_{N\to\infty} \frac{\lg N}{N} = \lim_{N\to\infty} \frac{1/N}{\ln 2} = \lim_{N\to\infty} \frac{1}{N \ln 2} = 0,$$

and thus we can assert that $\lg N$ is asymptotically genuinely smaller than N.

The Hierarchy of Orders of Magnitude (2)

Theorem 6.10. *For any $k > 0$ and for any $\varepsilon > 0$, we have*

$$\log^k x = o(x^\varepsilon).$$

Proof. What we need to show is that for any fixed k and ε, we have

$$\lim_{x\to\infty} \left| \frac{\log^k x}{x^\varepsilon} \right| = 0$$

We do this with L'Hôpital's rule, by differentiating numerator and denominator.

$$\begin{aligned}
\lim_{x\to\infty} \left| \frac{\log^k x}{x^\varepsilon} \right| &= \lim_{x\to\infty} \left| \frac{(k/x)\log^{k-1} x}{\varepsilon x^{\varepsilon-1}} \right| \\
&= \lim_{x\to\infty} \left| \frac{k\log^{k-1} x}{\varepsilon x^\varepsilon} \right| \\
&= \lim_{x\to\infty} \left| \frac{k(k-1)\log^{k-2} x}{\varepsilon^2 x^\varepsilon} \right| \\
&= \lim_{x\to\infty} \left| \frac{k!}{\varepsilon^k x^\varepsilon} \right| \\
&= 0
\end{aligned}$$

□

The Hierarchy – A Summary

• All the little oh truths imply the same as a big Oh truth, and transitivity implies that each of these subsumes the previous one.

- $\lg N = o(N^{1/2})$. Actually,
  - $\lg N = o(N^{1/2})$
  - $\lg N = o(N^{1/4})$
  - $\lg N = o(N^{1/8})$
  - and so forth.
- $N^{1/2} = o(N)$
- $N = o(N^{3/2})$
- $N^{3/2} = o(N^2)$
- $N^2 = o(N^{5/2})$
- $N^{5/2} = o(N^3)$

The Hierarchy – A Summary

- We also have the following.
  - $N \lg N = o(N^{3/2})$. Actually,
    - $\lg N = o(N^{3/2})$
    - $\lg N = o(N^{1+1/4})$
    - $\lg N = o(N^{1+1/8})$
    - $\lg N = o(N^{1+1/16})$
    - and so forth.
  - $N^{1/2} \lg N = o(N)$
  - $N \lg N = o(N^{3/2})$
  - $N^{3/2} \lg N = o(N^2)$
  - $N^2 \lg N = o(N^{5/2})$
  - $N^{5/2} \lg N = o(N^3)$

Big Theta

- Big Oh is like $\leq$
- Little oh is like $<$
- Big Theta is like "equals"

Definition 6.11. *We write*

$$f(x) = \Theta(g(x))$$

if there exist constants c, C_1, and C_2 such that for all $x > c$ we have

$$C_1 \cdot g(x) \leq |f(x)| \leq C_2 \cdot g(x).$$

Big Theta (2)

Theorem 6.12. *Let* $f(x) = a_k x^k + \cdots + a_0$. *Then* $f(x) = \Theta(x^k)$.

Proof. To do the formal proof, we need to show that there exists a constant C and a constant c such that for every $x > c$, we have $|f(x)| \leq C \cdot g(x)$. Let $C = (k+1)\max|a_i|$. We have

$$\begin{aligned}
|f(x)| &= |x^k \cdot (a_n + a_{k-1}/x + a_{k-2}/x^2 \cdots + a_0/x^k)| \\
&\leq |x^k| \cdot (|a_n| + |a_{k-1}/x| + |a_{k-2}/x^2| \cdots + |a_0/x^k|) \\
&\leq \max|a_i| \cdot |x^k| \cdot (|1| + |1/x| + |1/x^2| \cdots + |1/x^k|).
\end{aligned}$$

Now if $x > 1$, we have

$$\begin{aligned}
|f(x)| &\leq \max|a_i| \cdot x^k \cdot (1 + 1 + 1 \cdots + 1) \\
&= \max|a_i| \cdot x^k \cdot (k+1) \\
&= C \cdot x^k.
\end{aligned}$$

Therefore $f(x) = O(x^k)$.

Big Theta (3)

In the other direction, we need to show that there exists a constant C and a constant c such that for every $x > c$, we have $|f(x)| \geq C \cdot g(x)$. Let $C = \min |a_i|/k$. We have

$$\begin{aligned}
|f(x)| &= |x^k \cdot (a_n + a_{k-1}/x + a_{k-2}/x^2 \cdots + a_0/x^k)| \\
&\geq |x^k| \cdot (|a_n + a_{k-1}/x + a_{k-2}/x^2 \cdots| - |a_0/x^k|) \\
&\geq |x^k| \cdot (|a_n| - |a_{k-1}/x| - |a_{k-2}/x^2| \cdots - |a_0/x^k|) \\
&\geq \min |a_i| \cdot |x^k| \cdot (|1| - |1/x| - |1/x^2| \cdots - |1/x^k|).
\end{aligned}$$

Now if $x > k^2$, we have

$$\begin{aligned}
|f(x)| &\geq \min |a_i| \cdot x^k \cdot (1 - 1/k - 1/k^3 \cdots - 1/k^{2k-1}) \\
&\geq \min |a_i| \cdot x^k \cdot (1 - (k-1)/k) \\
&= \min |a_i| \cdot x^k/k \\
&= C \cdot x^k.
\end{aligned}$$

Therefore $f(x) = \Theta(x^k)$. □

Ignoring Fencepost Issues

- We don't like to have to count exactly correctly. (Is it N iterations, or $N+1$? What a pain!)
- Fortunately, we usually don't have to.

Theorem 6.13. *Let k be fixed and let $f(N) = (N+1)^k$ and $g(N) = (N-1)^k$. Then $f(N) = \Theta(N^k)$ and $g(N) = \Theta(N^k)$.*

Proof. Very crudely, we have for $N > 1$ that

$$(1/2)^k N^k = (N/2)^k < (N+1)^k < (2N)^k = 2^k N^k.$$

Since k is fixed, $(1/2)^k$ and 2^k are the constants needed for the $\Theta(\cdot)$.

A second version of a proof of this is also enlightening. We have

$$\begin{aligned} f(N) &= (N+1)^k \\ &= N^k + kN^{k-1} + \binom{k}{2} N^{k-2} + ... + \binom{k}{k-2} N^2 + kN^1 + 1 \end{aligned}$$

and now we can cite Theorem 6.6 to conclude that this polynomial is asymptotically exactly as large as its lead term.

□

Ignoring Fencepost Issues (2)

- If we did it once, we can do it twice. Or three times. Or any fixed number of times.

Theorem 6.14. *Let i and k be fixed and let $f(N) = (N + i)^k$. Then $f(N) = \Theta(N^k)$.*

Basic Orders of Magnitude Revisited

Theorem 6.15. *Let $f_a(N) = N^a$ and $f_b(N) = (\lg N)^b$ for any positive real numbers a and b. Then*

1. *If $0 < a_1 < a_2$, we have $f_{a_1}(N) = o(f_{a_2}(N))$.*

2. *If $0 < b_1 < b_2$, we have $f_{b_1}(N) = o(f_{b_2}(N))$.*

3. *If $0 < a_1 < a_2$ and if b_1 and b_2 are nonnegative, we have $f_{a_1}(N) \cdot f_{b_1}(N) = o(f_{a_2}(N) \cdot f_{b_2}(N))$.*

4. *If $0 < b_1 < b_2$, we have $f_a(N) \cdot f_{b_1}(N) = o(f_a(N) \cdot f_{b_2}(N))$ for any nonnegative values of a.*

Basic Orders of Magnitude Revisited (2)

Proof. To prove part 1, we consider

$$\lim_{x\to\infty} \frac{|f_{a_1}(x)|}{|f_{a_2}(x)|} = \lim_{x\to\infty} \frac{N^{a_1}}{N^{a_2}} = \lim_{x\to\infty} \frac{1}{N^{a_2-a_1}}.$$

By assumption, $a_2 - a_1 > 0$. The denominator of the above expression thus goes to infinity, which means that the expression must go to zero, which is exactly what is required to be covered by the definition of little oh.

Part 2 is entirely similar. We have

$$\lim_{x\to\infty} \frac{|f_{b_1}(x)|}{|f_{b_2}(x)|} = \lim_{x\to\infty} \frac{(\lg N)^{b_1}}{(\lg N)^{b_2}} = \lim_{x\to\infty} \frac{1}{(\lg N)^{b_2-b_1}}.$$

Again, the denominator of the above expression goes to infinity, which means that the expression must go to zero, which is what is required to be covered by the definition of little oh.

Basic Orders of Magnitude Revisited (3)

To prove part 3, we note that

$$\lim_{x\to\infty} \frac{N^{a_1}(\lg N)^{b_1}}{N^{a_2}(\lg N)^{b_2}} = \lim_{x\to\infty} \frac{(\lg N)^{b_1-b_2}}{N^{a_1-a_2}}.$$

If we have $b_1 - b_2 < 0$, then the log term belongs in the denominator, which goes to infinity, and the limit is clearly zero. If we have $b_1 - b_2 < 0$, then we have

$$\lim_{x\to\infty} \frac{(\lg N)^{b}}{N^{a}}$$

with $a > 0$ and $b > 0$, and by applying L'Hôpital's Rule once we get that this limit is zero.

Finally, proving part 4 is really the same as proving part 2 and cancelling some powers of N at the beginning.

□

Useful Corollaries

Corollary 6.16. *If $f(N) = \lg N$, then $f(N) = O(N^{\varepsilon})$ for any positive value of ε. In particular, we have then $f(N) = O(N^{1/2})$ and $f(N) = O(N)$.*

Corollary 6.17. *The restriction to exponent 1 in Corollary 6.16 is not necessary. In particular, if $f(N) = (\lg N)^a$ for any positive a, then $f(N) = o(N^{\varepsilon})$ for any positive value of ε, and then $f(N) = o(N^{1/2})$ and $f(N) = o(N)$.*

Corollary 6.18. *We can multiply by powers of N as we wish without affecting the asymptotics. In particular, if $f(N) = N \cdot (\lg N)^a$ for any positive a, then $f(N) = o(N^{1+\varepsilon})$ for any positive value of ε, and then $f(N) = o(N^{3/2})$ and $f(N) = o(N^2)$.*

Corollary 6.19. *More generally, if $f(N) = N^k \cdot (\lg N)^a$ for any positive a and k, then $f(N) = o(N^{k+\varepsilon})$ for any positive value of ε, and then $f(N) = o(N^{k+1/2})$ and $f(N) = o(N^{k+1})$.*

The Hierarchy Yet Again

- $(\lg N)^{1/2}$
- $\lg N$
- $(\lg N)^2$
- $N^{1/4}$
- $N^{1/2}$
- $N^{1/2} \lg N$
- $N^{1/2}(\lg N)^2$
- N
- $N \lg N$
- $N(\lg N)^2$
- $N^{3/2}$
- N^2

- $N^2 \lg N$
- N^3

Back to Code

- Bubblesort is *quadratic time*.
- We write *constant time* as $O(1)$.
- Linear time, assuming `something` is constant time

```
for(int i = 0; i < N-1; ++i)
{
  something
}
```

- Also linear time, assuming `something` is constant time

```
for(int i = 0; i < N-1; i = i+2)
{
  something
}
```

Back to Code (2)

```
for(int i = 0; i < N; ++i)
{
  for(int j = 0; j < N-1; j = j+2)
  {
    for(int k = 0; k < N/3; ++k)
    {
      binary search on an (N/2)-size subset of the array
    }
  }
}
```

- The outer loop clearly executes N times.
- The middle loop executes R times, where R is either $(N-1)/2$ if N is odd or $(N-2)/2$ if N is even.
- The inner loop executes S times, where S is one of $N/3$, $(N-1)/3$, or $(N-2)/3$.
- The inner computation takes $\lg(N/2)$ probes.

Back to Code (3)

We thus know that our program segment will require

$$N \cdot R \cdot S \cdot (\lg N - 1)$$

steps, where $R = N/2 - a$ with $a = 1/2$ or $a = 1$, and $S = N/3 - b$, with $b = 0$, $b = 1/3$, or $b = 2/3$.

If we blast out this product, we get

$$N \cdot R \cdot S \cdot (\lg N - 1) = N \cdot (N/2 - a) \cdot (N/3 - b) \cdot (\lg N - 1)$$

If we combine all the results from this chapter, we can argue that the first three factors in the above expression are each $\Theta(N)$ and the last factor is $\Theta(\lg N)$, so the product is $\Theta(N^3 \lg N)$, which is the overall running time of the triple loop with its inner binary search.

Make Sure You Know What You Are Counting

- Comparing two integers is constant time.
- Comparing two strings of fixed maximum length is constant time.
- Comparing two strings of *arbitrary* length is not constant time.

Starred Section – Exponential Orders of Magnitude

-
-
-
-

What Do We Count?

- Comparisons for sorting and searching.
- Multiplications and divisions in numerical computations.
- "Probes" into a data file if we are going into a big disk file?
- Do the bottleneck-ology.

Binary Divide-and-Conquer Is Optimal

• We have gone over the *divide and conquer* approach of binary search.

• This is important because it is optimal.

• In a divide-and-conquer algorithm, we would like to cut the problem space into two halves with each outer iteration, determine which half holds the answer, and then attack only that half, which is now a problem only half as big as what we had in the previous iteration.

Theorem 6.20. *Given a list L of N items in sorted order, binary search will determine if a potential value x is in L in $O(\lg N)$ comparisons. Binary search is optimal for searching in the sense that if any other search method has $O(f(N))$ running time, then $O(\lg N)) \leq O(f(N))$.*

Binary Divide-and-Conquer Is Optimal

Proof. We begin with an observation: It is impossible for a search algorithm to work correctly on all inputs unless a potential value x is compared against all N entries in the list L.

Clearly, if there is any element y in L that we do not compare to x, then we cannot determine whether $x = y$ or not and thus the algorithm cannot guarantee that the algorithm works correctly under all inputs. The algorithm must return an answer independent of the value of y, but both $y = x$ and $y \neq x$ are legal input data, so the algorithm cannot be right for all inputs.

It is necessary, therefore, that any search algorithm that actually works on all inputs must achieve the effect of at least N comparisons. What we now claim is that binary search achieves the effect of N comparisons by executing $O(\lg N)$ comparisons. But this is exactly what we discussed when we first presented binary search. With our first comparison, we are comparing against not just one element (the element at the midpoint of the list), but half the elements in the list. If x is found in the first comparison to lie in the first half of the list, then we have effectively compared x to all $N/2$ entries in the upper half of the list, since the "less-than" operator is transitive and the list is sorted. With our second comparison, we effectively compare against $N/4$

elements for the same reason. Since there are $\lg N$ summands on the right hand side of the equation

$$N = N/2 + N/4 + N/8 + ... + 1, \tag{1}$$

we can conclude that we have effectively made the necessary N comparisons although in truth we have only made $\lg N$ comparisons.[2]

At this point, then, we can conclude that we need N comparisons in effect and that we can accomplish that task with binary search using only $O(\lg N)$ actual comparisons. To complete the proof we must show that no other search method can run with fewer actual comparisons.

What we will show is that with every actual comparison that is made in binary search, we are making at least as much progress toward answering the search question as can be made by any other algorithm making any single comparison. That is, we observe that every comparison performed in binary search eliminates half the items in the sorted list, and no other algorithm can ever eliminate more than that many items when it makes a single actual comparison.

Consider the sorted list of N items. If we compare a potential element x against some specific element in the list, then that comparison splits the list L into one subset L_1 of elements $< x$ and one subset L_2

[2]We are going to ignore the issues of $<$ versus $\leq$ and of whether N is a perfect power of 2. If we really wanted to, we could be completely precise, but the precise proof will not offer any greater insight to the reader.

of elements $> x$. If we compare against the element at subscript k, then these two subsets are of size k and $N - k$, respectively, and regardless of the values of x and of k, there will be legal data input lists for which x is less than the k-th element and for which x is larger than the k-th element. In the worst case, then, we have made one comparison that has had the effect of $\min\{k, N - k\}$ comparisons and we have $\max\{k, N-k\}$ elements left in the list after one actual comparison. Since by choosing the midpoint in binary search the worst case is that we have $N/2$ elements, and by choosing any other subscript to test against we have $\max\{k, N-k\} \geq N/2$ remaining in the worst case, it is clear that the midpoint is the optimal choice for the first comparison. Comparisons after the first work in exactly the same way: with any actual comparison, we make at least as much progress with the midpoint choice of binary search than we can with any other choice, so we cannot do better overall than by performing binary search. □

Proving That an Algorithm is Optimal

• In general, proving something is possible is much easier than proving something is impossible.

• To prove that binary search was better than *any* other approach, we had to show that nothing would work better, not just that the other methods we could think of wouldn't work any better.

• First we determined the necessary cost of search under any circumstances, namely that the *effect* of N comparisons must be accomplished.

• Then we showed that binary search does what is necessary in its worst case running time.

• We showed that the progress made by one unit of work done by binary search was at least as good at every stage of the computation as the progress made by one unit of work for any other algorithm.

• If no single comparison can make more progress than the progress made in binary search with the corresponding comparison, then clearly no aggregate of comparisons can make more progress than the aggregate of comparisons in binary search.

Ternary Is No Better Than Binary

• If divide-and-conquer into equal halves is effective, then why not a ternary instead of a binary search?

• First, we note that we can in fact with two comparisons determine which third of the the array the potential entry x would fall in, by testing x against the elements at locations $N/3$ and $2N/3$. If we keep going, then instead of equation (1) with $\lg N$ summands, we have

$$N = 2N/3 + 2N/9 + 2N/27 + ... + 1, \tag{2}$$

with $\log_3 N$ summands. We can thus determine whether x is in the list with $2 \cdot \log_3 N$ comparisons. However, $2 \cdot \log_3 N = 2 \cdot \frac{\lg N}{\lg 3} \approx 1.26 \lg N$, so we are actually doing slightly more work than before. The decrease in the number of levels we have to search, from $\lg N$ to $\log_3 N$, is more than counterbalanced by the increased work at each level.

• And, of course, all logs are the same.

The End