Advanced sorting Algorithms

Runtime Analysis, part 2

January 2021

Thanks to Prof. Nurt Haspel for many of these slides

---

Runtime Analysis: from last class

When we develop an algorithm, we want to know how many resources it requires.

- Let T and N be positive numbers. N is the size of the problem, and T measures a resource: Runtime, CPU cycles, disk space, memory, etc.
- Order of growth can be important. For example – sorting algorithms can perform quadratically (n^2) or as n log(n).
- Very big difference for large inputs.
- We care less about constants, so 100N = O(N).
- 100N + 200 = O(N). Big-O expressions
- S&W use ~ for “tilde approximations”:

Here 100N + 200 ~ N. This means 100N + 200 = O(N) and 100N + 200 = Ω(N), a stronger statement, but the most important part is the big-O expression, the upper bound.

---

CS310 Class 2

Take roll for second and last time

Look at Linux setup for class at users.cs.umb.edu (or users1.cs.umb.edu)

Look at Linux setup for class — ~cs310 directory for your private cs310 files: mkdir hw1, cd hw1, ...

Class website: www.cs.umb.edu/cs310

--Available at /data/htdocs/cs310 in the filesystem

ssh users.cs.umb.edu, login
cd /data/htdocs/cs310, ls, is —lr to see files

---

Useful Examples

Function Name

| O   | Constant |
| log N | Logarithmic |
| log^2 N | Log-squared |
| N   | Linear |
| N log N | Linearithmic |
| N^2 | Quadratic |
| N^3 | Cubic |
| 2^N | Exponential |

(N log N is called “linearithmic”, page 187)

---

Runtime Table

<table>
<thead>
<tr>
<th>n</th>
<th>1(n)</th>
<th>2n</th>
<th>100n</th>
<th>10,000n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.001µs</td>
<td>0.001µs</td>
<td>0.001µs</td>
<td>0.001µs</td>
</tr>
<tr>
<td>10</td>
<td>0.001µs</td>
<td>0.001µs</td>
<td>0.001µs</td>
<td>0.001µs</td>
</tr>
<tr>
<td>100</td>
<td>0.001µs</td>
<td>0.001µs</td>
<td>0.001µs</td>
<td>0.001µs</td>
</tr>
<tr>
<td>1000</td>
<td>0.001µs</td>
<td>0.001µs</td>
<td>0.001µs</td>
<td>0.001µs</td>
</tr>
</tbody>
</table>

(N log N is called “linearithmic”, page 187)
Adding and Multiplying Functions

- Rule for sums (e.g. two consecutive blocks of code): If \( T_1(N) = O(F(N)) \) and \( T_2(N) = O(G(N)) \) then \( T_1 + T_2 = O(\max(F(N), G(N))) \). The biggest contribution dominates the sum. Same for \( n \) here and below.
- Rule for products (e.g. an inner loop run by an outer loop): If \( T_1(N) = O(F(N)) \) and \( T_2(N) = O(G(N)) \) then \( T_1 \cdot T_2 = O(F(N) \cdot G(N)) \).
- Example: \( (n^2 + 2n + 17) \cdot (2n^2 + n + 17) = O(n^2 \cdot n^2) = O(n^4) \).
  (Remember to ignore all but the leading term.)
- If we sum over a large number of terms, we multiply the number of terms by the estimated size of one term.
- Example: Sum of \( i \) from 1 to \( N \). Average size of an element: \( \frac{N}{2} \). There are \( N \) terms so the sum is \( O(N^2) \). Exact form: \( \frac{N(N+1)}{2} \).

Loops

- The runtime of a loop is the runtime of the statements in the loop * number of iterations.
- Example: bubble sort

```c
/* sort array of ints in A[0] to A[n-1] */
int bubblesort(int A[], int n) {
    int i, j, temp;
    for(i = 0; i < n-1; i++) /* n passes of loop */
        for(j = n-1; j > i; --j) /* n-i passes of loop */
            if (A[j-1] > A[j]) { // all these are O(1)
                temp = A[j-1];   // O(1) body
                A[j] = temp;
            }
}
```

Array accesses

- This code has many array element references
- What is the performance of one array ref, say \( A[k] \) where \( k \) is a particular number like 10345?
- You might worry that this involves significant work to get out to element 10345 from the start of the array.
- But in fact the CPU can compute where the element is from the address of \( A \) and this number:
  - Element addr = \( A + 8^*10345 \) if the element is 8 bytes long.
  - This calculation is just one multiplication and one add, so \( O(1) \) in all. Then the memory access is also \( O(1) \).

Loops

The plan: work from inside out:

- Calculate the body of inner loop (constant -- an if statement
  and three assignments using \( A[] \) refs).
- Estimate the number of passes of the inner loop: \( n-i \) passes.
- Work out the Big O of the inner loop
- Estimate the number of passes of the outer loop: \( n \) passes.
- Work out the Big O of the outer loop, i.e. the whole thing

```c
/* sort array of ints in A[0] to A[n-1] */
int bubblesort(int A[], int n) {
    int i, j, temp;
    for(i = 0; i < n-1; i++) /* n passes of loop */
        for(j = n-1; j > i; --j) /* n-i passes of loop */
            if (A[j-1] > A[j]) { // all these are O(1)
                temp = A[j-1];   // O(1) body
                A[j] = temp;
            }
}
```

Loops

The plan: work from inside out, outer loop part:

- We now know the inner loop takes \( O(n-i) \)
- Estimate the number of passes of the outer loop: \( n \) passes.
- So the outer loop takes \( O(n) + O(n-1) + O(n-2) + \ldots + O(1) \)
  - \( = O(n^2/2) = \Theta(n^2) \)

```c
/* sort array of ints in A[0] to A[n-1] */
int bubblesort(int A[], int n) {
    int i, j, temp;
    for(i = 0; i < n-1; i++) /* n passes of loop */
        for(j = n-1; j > i; --j) /* n-i passes of loop */
            if (A[j-1] > A[j]) { // all these are O(1)
                temp = A[j-1];   // O(1) body
                A[j] = temp;
            }
}
```
Recursive Functions

- Recursive functions perform some operations and then call themselves with a different (usually smaller) input.
- Example: factorial.

```c
int factorial (int n)
{
    if (n==1) return 1;
    return n*factorial(n-1);
}
```

Recursive Analysis

- Let us denote the run-time on input $n$ as some function $T(n)$ and analyze $T(n)$.
- $O(1)$ operations before recursive call – if statement and a multiplication.
- The recursive part calls the same function with $n-1$ as input, so this part runs $T(n-1)$.
- So: $T(n) = c + T(n-1)$.
- Similarly: $T(n-1) = c + T(n-2)$.
- $T(n) = (n-1)c + k = O(n)$.
- Iterative performs the same.

A Problematic Example

- The well-known Fibonacci series, where each number is the sum of the previous two numbers: 0 1 1 2 3 5 8 13 ...
- $f(n) = f(n-1) + f(n-2)$, where $f(0) = 0, f(1) = 1$
- This is a recursive definition.
- The following recursive program calculates the $n^{th}$ term in the Fibonacci series (assume $n$ is non-negative and the first term is the zero-th):

```c
int fib(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-2)+fib(n-1);
}
```

Ill-Behaved Recursion

- The problem is the double recursion which runs on the same input so adds a lot of redundant work.
- The call tree looks like a big binary tree.
- Two or more recursive calls are not necessarily bad, as long as we split the work too!
- Example: Merge sort – sort recursively two halves of an array and merge.
- Call recursively twice, but on different input! The work is split between recursive calls in a smart way.
- The exact runtime of `fib` is $O(1.618^n)$. The full analysis is beyond the scope for now.
- But it is exponential! (remember the illustration above).
- How do we fix the fibonacci program? See geeksforgeeks.
**Binary Search**

- Definition: Search for an element in a sorted array.
- Return array index where element is found or a negative value if not found.
- Implemented in Java as part of the Collections API.
- Start in the middle of the array.
- If the element is smaller than that, search in the smaller half. Otherwise – search in the larger half.

**Binary Search Example**

<table>
<thead>
<tr>
<th>Key</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>&gt;4</td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>&lt;8</td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

**Binary Search Implementation**

```java
private static int binarySearch(int[] a, int low, int high, int x)
{
    if (low > high)
        return NOT_FOUND;
    int mid = (low + high) / 2;
    if (a[mid] < x)
        return binarySearch(a, mid + 1, high, x);
    else if (a[mid] > x)
        return binarySearch(a, low, mid - 1, x);
    else
        return mid;
}
```

**Binary Search Algorithm Runtime**

You should be able to guess this one out by now (I hope):

For case of size N, we need to do recursive call on N/2, plus a little more work:

\[ T(N) = T\left(\frac{N}{2}\right) + O(1) = T\left(\frac{N}{2}\right) + c \]

\[ T\left(\frac{N}{2}\right) = T\left(\frac{N}{4}\right) + c + c = T\left(\frac{N}{4}\right) + c + c \]

= \ldots \log(N) divisions by 2 to get to T(1), each one with + c

\[ T(N) = O(\log N) \]

**Mergesort Recurrence Formula**

\[ T(n) = \begin{cases} C & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{otherwise} \end{cases} \]

Notice that c and C are not the same constant!

**Mergesort Recurrence Formula**

- Identities like this come up frequently in algorithmic analysis.
- It’s important to have ways of solving them. We’ll see a couple.
- One basic way is to form a recursion tree.
- Mergesort is a good example:
  - If the array has at most one item – return.
  - Split it in half, call merge sort recursively on each half.
  - Merge the two sorted halves.
Elements to be sorted

For merge sort, and other sorts, we need to compare elements of the array, for example x.compareTo(y).

Thus the elements need to be “Comparable” (technically, implementing the Comparable interface).

This means the elements have a compareTo method that follows the expected rules. See page 247.

They can have lots of other methods too. See the Date class on page 247 that has a compareTo method.

We see that Date objects are Comparable, and thus can be sorted by merge sort and other sorts.

Explanation of fix

Previously on this last slide, the tmpArray was created by

```
Comparable[] tmpArray = new Comparable[a.length];
```

But we can’t use `new` with an interface type like Comparable, only with concrete classes.

We can copy an array with Arrays.copyOf, and this will give us the working space we need:

```
Comparable[] tmpArray = Arrays.copyOf(a, a.length);
```

This creates the array object and copies the element references from a to tmpArray (“shallow copy”).

The Mergesort Algorithm, like S&W page 273

```java
public static void mergeSort(Comparable[] a)
{
    Comparable[] tmpArray = Arrays.copyOf(a, a.length);  
    // see next slide for explanation
    mergeSort(a, tmpArray, 0, a.length - 1);
}
```

// Internal method that makes recursive calls
private static void mergeSort(Comparable[] a, Comparable[] tmpArray, int left, int right)
{
    if (left < right) {
        int center = (left + right) / 2;
        mergeSort(a, tmpArray, left, center);
        mergeSort(a, tmpArray, center + 1, right);
        merge(a, tmpArray, left, center + 1, right);
    }
}
```

Internal Merge Method

- For code for merge, see page 271, and add an argument to allow the tmpArray to be passed in.
  - (The code on page 271 is meant to be in a class that has a private static instance of array of Comparable named aux. See page 273)

- The basic idea of merge() is to “zipper” the two sorted arrays together, inching down both arrays together picking the next one for the output array.

- Let’s look at an example...

MergeSort Performance

Each pass of size N requires 2 calls on N/2 elements plus O(N) work to merge those two parts...

T(N) = 2 + T(N/2) + O(N)
    = 2 + (2 + T(N/4) + O(N/2)) + O(N)
    = 4 + T(N/4) + O(N)
    = 4 + (2 + T(N/8) + O(N/4)) + O(N) + O(N)
    = 8 + T(N/8) + O(N) + O(N) + O(N)
    = …… = 2^logN + T(1) + O(N) + O(N) + … + O(N)
    = N + O(1) + O(N) + O(N) + … + O(N).

The O(N) terms at the end are repeated log N times. log N terms of O(N) = O(Nlog N)

That’s a good sort!
**Recursion Tree for Merge Sort**

- If \( N = 2^n \) then there are \( p \) rows in the call hierarchy.
  In other words, \( p = \text{height of tree} = \log(N) \).

- Although this is double recursion, it is well-behaved because the work is cut in half in the recursive call (and there is not too much additional work).

---

**Another Way to Look at Runtime**

- What does “linear runtime” really mean?
- A linear function in math is \( f(x) = ax + b \), but for us, the \( b \) part is negligible compared to the \( ax \) part, so “linear runtime” means \( T(N) = O(N) \).
- Say a linear algorithm runs for 5 seconds on an input of size 10. How much time will it (approximately) run on an input of size 20?

---

**Another Way to Look at Runtime: Linear Case**

- Say a linear algorithm runs for 5 seconds on an input of size 10. How much time will it (approximately) run on an input of size 20?
- Answer: 10 seconds. Proof:
  \[ f(n) = O(n) \Rightarrow f(n) = cn \text{ for some } c \text{, approximately. } \]
  \[ f(2n) = c \cdot 2n = 2^1 f(n) = \text{twice the time for } n \]

- Doubling the input size roughly doubles the runtime.
- Similarly, increasing the input by a multiple of 10 increases the runtime by a factor or 10, in the linear algorithm case.

- If a quadratic algorithm \( f(n) = O(n^2) \) runs for 5 seconds on an input of size 10. How much time will it (approximately) run on an input of size 20?

---

**Another Way to Look at Runtime: quadratic case**

- If a quadratic algorithm \( f(n) = O(n^2) \) runs for 5 seconds on an input of size 10. How much time will it (approximately) run on an input of size 20?
- Answer: 4 times as much: 4 * 5 = 20 seconds. Proof:
  \[ f(n) = O(n^2) \Rightarrow f(n) = c \cdot n^2 \text{ for some } c. \text{ This means } \]
  \[ f(2n) = c \cdot (2n)^2 = 4c \cdot n^2 = 4 \cdot f(n) = 4 \text{ times the time} \]

- Doubling the input size roughly quadruples the runtime.
- Similarly, increasing the input by a multiple of 10 increases the runtime by a factor of 100, in the quadratic algorithm case.

- If an exponential algorithm \( f(n) = O(\exp(n)) \) runs for 5 seconds on an input of size 10. How much time will it (approximately) run on an input of size 20?

---

**Another Way to Look at Runtime: exponential case**

- If an exponential algorithm with \( T(N) = O(2^N) \) runs 5 seconds on an input of size 10. How much time will it (approximately) run on an input of size 20?
- Answer: \( 2^{10} = 1024 \) seconds = 1.4 hours. Proof:
  \[ f(N) = O(2^N) \Rightarrow f(N) = c \cdot 2^N \text{ for some } c. \text{ This means } \]
  \[ f(2N) = c \cdot 2^{2N} = c \cdot 4^N = 2^2 \cdot 2^N = 2^N \cdot f(n) = 2^N \text{ times the time} \]

- For every unit increase in input, we expect the time to double, so 10 units increase means \( 2^{10} \text{ times the time} \text{: very fast growth!} \)

- If an logarithmic algorithm \( f(n) = O(\log(n)) \) runs for 5 seconds on an input of size 10. How much time will it (approximately) run on an input of size 20?

---

**Another Way to Look at Runtime: logarithmic case**

- If an logarithmic algorithm \( f(n) = O(\log(n)) \) runs for 5 seconds on an input of size 10. How much time will it (approximately) run on an input of size 20?
- Answer: This is tricky. No simple rule
  \[ f(N) = O(\log N) \Rightarrow f(N) = c \cdot \log N \text{ for some } c. \text{ This means } \]
  \[ f(2N) = c \cdot \log(2N) = c \cdot \log 2 + \log N = f(N) + c \cdot \log 2. \]
  But what is \( c \) here? Our only clue is the original time:
  \[ 5 = c \cdot \log 10, \text{ so } c \text{ is approximately } \frac{5}{\log 10}. \text{ Thus } f(2N) = f(n) + 5 \cdot \log(2) \cdot \log 10 = f(n) + 5 \cdot \frac{5}{\log 10} \cdot f(n) = f(n) + 1.5 \times f(n) \]

- For every doubling in input, we expect the time in this case to increase by 1.5 units, so input of size 20 instead of 10 should take approximately \( 5 + 1.5 = 6.5 \) seconds.

Again we see how logarithmic growth is very slow.