3. DIVIDE-AND-CONQUER

Raveen de Silva, r.desilva@unsw.edu.au
office: K17 202
Course Admin: Anahita Namvar, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 2, 2022

Table of Contents

Table of Contents

An old puzzle

Problem

We are given 27 coins of the same denomination; we know that one of them is counterfeit and that it is lighter than the others. Find the counterfeit coin by weighing coins on a pan balance only three times.

Hint

You can reduce the search space by a third in one weighing!

An old puzzle

Solution

- Divide the coins into three groups of nine, say A, B and C.
- Weigh group A against group B.
  - If one group is lighter than the other, it contains the counterfeit coin.
  - If instead both groups have equal weight, then group C contains the counterfeit coin!
- Repeat with three groups of three, then three groups of one.

Divide and Conquer

- This method is called "divide-and-conquer".
- We have already seen a prototypical "serious" algorithm designed using such a method: the MERGE-SORT.
- We split the array into two, sort the two parts recursively and then merge the two sorted arrays.
- We now look at a closely related but more interesting problem of counting inversions in an array.

Table of Contents

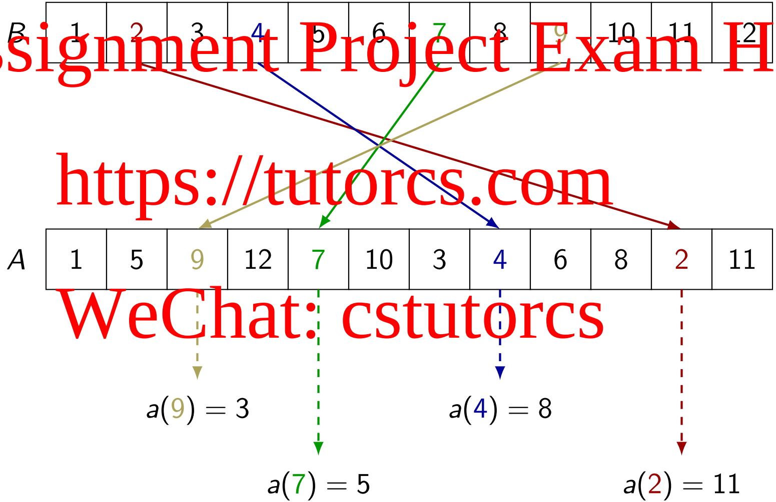
Counting the number of inversions

- Assume that you have m users ranking the same set of n movies. You want to determine for any two users A and B how similar their tastes are (for example, in order to make a recommender system).

- How should we measure the degree of similarity of two users A and B?

- Let's enumerate the movies on the ranking list of user B by assigning to the top choice of user B index 1, assign to his second choice index 2 and so on.
- For the i^{th} movie on B's list we can now look at the index of that movie on A's list, denoted by $a(i)$.

Counting the number of inversions



Counting the number of inversions

A good measure of how different these two users are is the number of pairs of movies which are 'out of order' between the two lists.

Definition

An *inversion* is a pair (i, j) such that:

- $i < j$, i.e. B prefers i to j, and
- $a(i) > a(j)$, i.e. A prefers j to i.

Our task will be to count the total number of inversions.

Counting the number of inversions

B	1	2	3	4	5	6	7	8	9	10	11	12

A	1	5	9	12	7	10	3	4	6	8	2	11

For example, 1 and 2 do not form an inversion because

$$1 = a(1) < a(2) = 11,$$

but 4 and 7 do form an inversion because

$$5 = a(7) < a(4) = 8.$$

Counting the number of inversions

- A brute force algorithm to count the inversions is to test each pair $i < j$, and add one to the total if $a(i) > a(j)$. Unfortunately this produces a quadratic time algorithm, $T(n) = \Theta(n^2)$.

- We now show that this can be done more efficiently, in time $O(n \log n)$, by applying a divide-and-conquer strategy.

- Clearly, since the total number of pairs is quadratic in n, we cannot afford to count the inversions one-by-one.
- The main idea is to tweak the MERGE-SORT algorithm, by extending it to recursively both sort an array A **and** determine the number of inversions in A.

Counting the number of inversions

- We split the array A into two (approximately) equal parts $A_{lo} = A[1..m]$ and $A_{hi} = A[m+1..n]$, where $m = \lfloor n/2 \rfloor$.

- Note that the total number of inversions in array A is equal to the sum of the number of inversions $I(A_{lo})$ in A_{lo} (such as 9 and 7) plus the number of inversions $I(A_{hi})$ in A_{hi} (such as 4 and 2) plus the number of inversions $I(A_{lo}, A_{hi})$ across the two halves (such as 7 and 4).

$$A \quad \underbrace{\boxed{1}\boxed{5}\boxed{9}\boxed{12}\boxed{7}\boxed{10}}_{A_{lo}}\ \underbrace{\boxed{3}\boxed{4}\boxed{6}\boxed{8}\boxed{2}\boxed{11}}_{A_{hi}}$$

Counting the number of inversions

- We have

$$I(A) = I(A_{lo}) + I(A_{hi}) + I(A_{lo}, A_{hi}).$$

- The first two terms of the right-hand side are the number of inversions within A_{lo} and within A_{hi}, which can be calculated recursively.

- It seems that the main challenge is to evaluate the last time, which requires us to count the inversions which cross the partition between the two sub-arrays.

Counting the number of inversions

- In our example, how many inversions involve the 8?

$$A \quad \underbrace{\boxed{1}\boxed{5}\boxed{9}\boxed{12}\boxed{7}\boxed{10}}_{A_{lo}}\ \underbrace{\boxed{3}\boxed{4}\boxed{6}\boxed{8}\boxed{2}\boxed{11}}_{A_{hi}}$$

- It's the number of elements of A_{lo} which are greater than 8.

- How would one compute this systematically?

Counting the number of inversions

- The idea is to not only count inversions across the partition, but also sort the array. We can then assume that the subarrays A_{lo} and A_{hi} are sorted in the process of counting $I(A_{lo})$ and $I(A_{hi})$.

- We proceed to count $I(A_{lo}, A_{hi})$ (specifically, counting each inversion according to the lesser of its elements) and simultaneously merge as in MERGE-SORT.

Counting the number of inversions

Each time we reach an element of A_{hi}, we have inversions between this number and each of the remaining elements in A_{lo}. We therefore add the number of elements remaining in A_{lo} to the answer.

A_{lo}					
1	5	7	9	10	12

A_{hi}					
2	3	4	6	8	11

A											
1	2	3	4	5	6	7	8				

$$\text{count} = 5 + 5 + 5 + 4 + 3 + 1 = 23.$$

Counting the number of inversions

- On the other hand, when we reach an element of A_{lo}, all inversions involving this number have already been counted.

- We have therefore counted the number of inversions within each subarray ($I(A_{lo})$ and $I(A_{hi})$) recursively as well as the number of inversions across the partition ($(I(A_{lo}, A_{hi}))$, and adding these gives $I(A)$ as required.

- Our modified Merge still takes linear time, so this algorithm has the same complexity as Merge-Sort, i.e. $\Theta(n \log n)$.
- **Next:** we look to generalise the divide and conquer method.

Table of Contents

Table of Contents

Recurrences

- Recurrences are important to us because they arise in estimations of time complexity of divide-and-conquer algorithms.
- Recall that counting inversions in an array A of size n required us to:
  - recurse on each half of the array (A_{lo} and A_{hi}), and
  - count inversions across the partition, in linear time.
- Therefore the runtime $T(n)$ satisfies

$$T(n) = 2T\left(\frac{n}{2}\right) + c\,n.$$

Recurrences

- Let $a \geq 1$ be an integer and $b > 1$ a real number, and suppose that a divide-and-conquer algorithm:

  - reduces a problem of size n to a many problems of smaller size n/b;
  - with overhead cost of $f(n)$ to split up the problem and combine the solutions from these smaller problems.

- The time complexity of such an algorithm satisfies

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n).$$

Recurrences

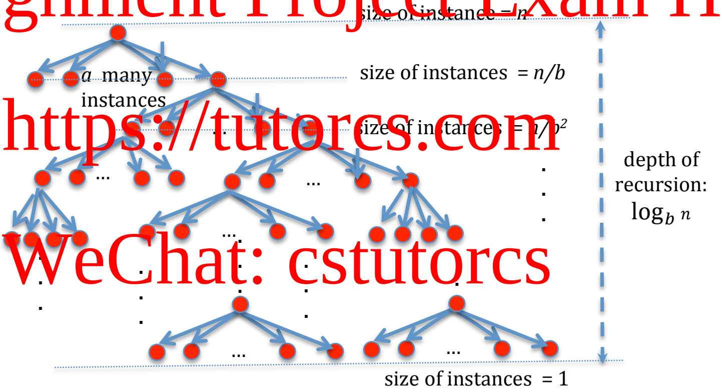
Note

Technically, we should be writing

$$T(n) = a\,T\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n)$$

but it can be shown that the same asymptotics are achieved if we ignore the rounding and additive constants.

Recurrences of the form $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$



- Some recurrences can be solved explicitly, but this tends to be tricky.

- Fortunately, to estimate the efficiency of an algorithm we **do not** need the exact solution of a recurrence.

- We only need to find:
  - the **growth rate** of the solution i.e., its asymptotic behaviour, and
  - the (approximate) **sizes of the constants** involved (more about that later)

- This is what the **Master Theorem** provides (when it is applicable).

Table of Contents

Master Theorem

Setup

Let:

- $a \geq 1$ be an integer and and $b > 1$ be a real number;

- $f(n) \geq 0$ be a non-decreasing function defined on the positive integers;

- $T(n)$ be the solution of the recurrence

$$T(n) = a\,T(n/b) + f(n).$$

Define the *critical exponent* $c^* = \log_b a$ and the *critical polynomial* n^{c^*}.

Master Theorem

Theorem

1. If $f(n) = O(n^{c^*-\varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{c^*})$;

2. If $f(n) = \Theta(n^{c^*})$, then $T(n) = \Theta(n^{c^*} \log n)$;

3. If $f(n) = \Omega(n^{c^*+\varepsilon})$ for some $\varepsilon > 0$, **and** for some $c < 1$ and some n_0,
$$a\, f(n/b) \leq c\, f(n) \tag{1}$$
holds for all $n > n_0$, then $T(n) = \Theta(f(n))$;

Exercise

Prove that $f(n) = \Omega(n^{c^*+\varepsilon})$ is a consequence of (1).

Master Theorem

Theorem (continued)

4. If none of these conditions hold, the Master Theorem is NOT applicable.

   Often, the proof of the Master Theorem can be tweaked to (asymptotically) solve such recurrences anyway! An example is $T(n) = 2T(n/2) + n \log n$.

Master Theorem

Remark

- Recall that for $a, b > 1$, $\log_a n = \Theta(\log_b n)$, so we can omit the base and simply write statements of the form $f(n) = \Theta(g(n)\log n)$.

- However, $n^{\log_a x}$ is not interchangeable with $n^{\log_b x}$ - the base must be specified in such expressions.

Master Theorem

Example 1

Let $T(n) = 4\,T(n/2) + n$.

Then the critical exponent is $c^* = \log_b a = \log_2 4 = 2$, so the critical polynomial is n^2.

Now, $f(n) = n = O(n^{2-\varepsilon})$ for small ε (e.g. 0.1).

This satisfies the condition for case 1, so $T(n) = \Theta(n^2)$.

Master Theorem

Example 2

Let $T(n) = 2T(n/2) + 5n$.

Then the critical exponent is $c^* = \log_b a = \log_2 2 = 1$, so the critical polynomial is n.

Now, $f(n) = 5n = \Theta(n)$.

This satisfies the condition for case 2, so $T(n) = \Theta(n \log n)$.

Master Theorem

Example 3

Let $T(n) = 3\,T(n/4) + n$.

Then the critical exponent is $c^* = \log_4 3 \approx 0.7925$, so the critical polynomial is $n^{\log_4 3}$.

Now, $f(n) = n = \Omega(n^{\log_4 3+\varepsilon})$ for small ε (e.g. 0.1).

Also, $af(n/b) = 3f(n/4) = 3/4\ n\ < c\,n = cf(n)$ for $c = .9 < 1$.

This satisfies the condition for case 3, so $T(n) = \Theta(f(n)) = \Theta(n)$.

Master Theorem

Example 4

Let $T(n) = 2\,T(n/2) + n \log_2 n$.

Then the critical exponent is $c^* = \log_2 2 = 1$, so the critical polynomial is n.

Now, $f(n) = n \log_2 n = \omega(n)$, so the conditions for case 1 and 2 do not apply.

However,

$$f(n) \neq \Omega(n^{1+\varepsilon}), \tag{2}$$

no matter how small we choose $\varepsilon > 0$.

Therefore the Master Theorem does **not** apply!

Master Theorem

Exercise

Prove (2), that is, for all $\varepsilon > 0$, $c > 0$ and $N > 0$ there is some $n > N$ such that

$$\log_2 n < c \cdot n^{\varepsilon}.$$

Hint

Use *L'Hôpital's rule* to show that

$$\frac{\log n}{n^{\varepsilon}} \to 0.$$

Master Theorem

Suppose $T(n)$ satisfies the recurrence

$$T(n) = a\left[T\left(\frac{n}{b}\right)\right] + f(n) \tag{3}$$

However, the $T(n/b)$ term can itself be reduced using the recurrence as follows:

$$T\left(\frac{n}{b}\right) = a\,T\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)$$

Substituting into (3) and simplifying gives

$$\begin{aligned} T(n) &= a\left[a T\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right] + f(n) \\ &= a^2 T\left(\frac{n}{b^2}\right) + a\,f\left(\frac{n}{b}\right) + f(n). \end{aligned}$$

Master Theorem

We have now established

$$T(n) = a^2 \left[T\left(\frac{n}{b^2}\right) \right] + a\, f\left(\frac{n}{b}\right) + f(n). \qquad (4)$$

But why stop there? We can now reduce the $T\left(n/b^2\right)$ term, again using (3):

$$T\left(\frac{n}{b^2}\right) = a\, T\left(\frac{n}{b^3}\right) + f\left(\frac{n}{b^2}\right).$$

We now substitute this into (4) and simplify to get

$$\begin{aligned} T(n) &= a^2 \left[a T\left(\frac{n}{b^3}\right) + f\left(\frac{n}{b^2}\right) \right] + a\, f\left(\frac{n}{b}\right) + f(n) \\ &= a^3\, T\left(\frac{n}{b^3}\right) + a^2\, f\left(\frac{n}{b^2}\right) + a\, f\left(\frac{n}{b}\right) + f(n). \end{aligned}$$

We can see a pattern emerging!

Master Theorem

Continuing in this way, we find that

$$T(n) = a^k\, T\left(\frac{n}{b^k}\right) + a^{k-1}\, f\left(\frac{n}{b^{k-1}}\right) + \ldots + a\, f\left(\frac{n}{b}\right) + f(n)$$

$$= a^k\, T\left(\frac{n}{b^k}\right) + \sum_{i=0}^{k-1} a^i\, f\left(\frac{n}{b^i}\right).$$

We stop when $k = \lfloor \log_b n \rfloor$, since this gives $n/b^k \approx 1$.

$$T(n) \approx a^{\log_b n}\, T\left(\frac{n}{b^{\log_b n}}\right) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i\, f\left(\frac{n}{b^i}\right).$$

Master Theorem

Now we have

$$T(n) \approx a^{\log_b n}\, T\left(\frac{n}{b^{\log_b n}}\right) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right).$$

We can use the identity $a^{\log_b n} = n^{\log_b a}$ to get:

$$T(n) \approx n^{\log_b a}\, T(1) + \underbrace{\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right)}_{S}. \tag{5}$$

Importantly, we have not assumed anything about $f(n)$ yet! We will now analyse the sum S in the simplest case of the Master Theorem, namely Case 2.

Master Theorem: Case 2

Suppose $f(n) = \Theta\left(n^{\log_b a}\right)$. Then

$$\begin{aligned} S &= \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) \\ &= \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i \,\Theta\left(\left(\frac{n}{b^i}\right)^{\log_b a}\right) \\ &= \Theta\left(\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a}\right), \end{aligned}$$

using the sum property and scalar multiple property.

Master Theorem: Case 2

$$S = \Theta\left(\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a}\right)$$

$$= \Theta\left(n^{\log_b a} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i \left(\frac{1}{b^i}\right)^{\log_b a}\right)$$

as $n^{\log_b a}$ is common to every term of the sum,

$$= \Theta\left(n^{\log_b a} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(\frac{a}{b^{\log_b a}}\right)^i\right).$$

Master Theorem: Case 2

$$S = \Theta\left(n^{\log_b a} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(\frac{a}{b^{\log_b a}}\right)^i\right)$$

$$= \Theta\left(n^{\log_b a} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(\frac{a}{a}\right)^i\right)$$

as $b^{\log_b a} = a$,

$$= \Theta\left(n^{\log_b a} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} 1\right)$$

$$= \Theta\left(n^{\log_b a} \lfloor \log_b n \rfloor\right).$$

Master Theorem: Case 2

Finally, we return to (5). Substituting our result for S gives

$$
\begin{aligned}
T(n) &= n^{\log_b a} T(1) + \Theta\left(n^{\log_b a} \lfloor \log_b n \rfloor\right) \\
&= \Theta\left(n^{\log_b a} \log n\right)
\end{aligned}
$$

as logarithms of any base are equivalent,

completing the proof.

Master Theorem: Other cases

- The proof of Case 1 is very similar to the above. The main difference is that $\sum 1$ is replaced by $\sum (b^{\varepsilon})^i$, forming a geometric series, which can be summed using the identity

$$1 + r + r^2 + \ldots + r^{k-1} = \frac{r^k - 1}{r - 1}.$$

- In Case 3, we need to prove that $T(n) = \Theta(f(n))$, that is, both:
  - $T(n) = \Omega(f(n))$, which follows directly from the recurrence $T(n) = a\,T(n/b) + f(n)$, and
  - $T(n) = O(f(n))$ (not as obvious).

Master Theorem: Case 3

Exercise

Prove that in Case 3, $T(n) = O(f(n))$.

Hint

You will need to bound

$$S = \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right)$$

from above. Try using the inequality

$$a\, f\left(\frac{n}{b}\right) \le c\, f(n)$$

to relate each term of S to $f(n)$.

Table of Contents

Basics revisited: how do we add two integers?


```
 C C C C C      carry
   X X X X X    first integer
+  X X X X X    second integer
  -----------
 X X X X X X    result
```


- Adding 3 bits can be done in constant time.
- It follows that the whole algorithm runs in linear time i.e., $O(n)$ many steps.

Basics revisited: how do we add two integers?

Question

Can we add two n-bit numbers in faster than in linear time?

Answer

No! There is no asymptotically faster algorithm because we have to read every bit of the input, which takes $O(n)$ time.

Basics revisited: how do we multiply two integers?

```
    X X X X  <- first input integer
  * X X X X  <- second input integer
    -------
    X X X X  \
  X X X X     \ O(n^2) intermediate operations:
X X X X       / O(n^2) elementary multiplications
X X X X      /    + O(n^2) elementary additions
---------------
X X X X X X X X  <- result of length 2n
```

- We assume that two `X`'s can be multiplied in $O(1)$ time (each `X` could be a bit or a digit in some other base).
- Thus the above procedure runs in time $O(n^2)$.

Basics revisited: how do we multiply two integers?

Question

Can we multiply two n-bit numbers in linear time, like addition?

Answer

No one knows! "Simple" problems can actually turn out to be difficult!

Question

Can we do it in faster than quadratic time? Let's try divide and conquer.

Table of Contents

Multiplying large integers by divide-and-conquer

- Split the two input numbers A and B into halves:
  - A_0, B_0 - the least significant $n/2$ bits;
  - A_1, B_1 - the most significant $n/2$ bits.

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- AB can now be calculated recursively using the following equation:

$$AB = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{\frac{n}{2}} + A_0 B_0.$$

Multiplying large integers by divide-and-conquer

```
function MULT(A, B)
    if |A| = |B| = 1 then return AB
    else
        A_1 ← MoreSignificantPart(A);
        A_0 ← LessSignificantPart(A);
        B_1 ← MoreSignificantPart(B);
        B_0 ← LessSignificantPart(B);
        X ← MULT(A_0, B_0);
        Y ← MULT(A_0, B_1);
        Z ← MULT(A_1, B_0);
        W ← MULT(A_1, B_1);
        return W 2^n + (Y + Z) 2^{n/2} + X
    end if
end function
```

Multiplying large integers by divide-and-conquer

How many steps does this algorithm take?

Each multiplication of two n digit numbers is replaced by four multiplications of $n/2$ digit numbers: A_1B_1, A_1B_0, B_1A_0, A_0B_0, plus we have a **linear** overhead to shift and add:

$$T(n) = 4T\left(\frac{n}{2}\right) + c\,n.$$

Let's use the Master Theorem!

Multiplying large integers by divide-and-conquer

$$T(n) = 4T\left(\frac{n}{2}\right) + c\,n$$

The critical exponent is $c^* = \log_2 4 = 2$, so the critical polynomial is n^2.

Then $f(n) = c\,n = O(n^{2-0.1})$, so Case 1 applies.

We conclude that $T(n) = \Theta(n^{c^*}) = \Theta(n^2)$, i.e., we gained **nothing** with our divide-and-conquer!

Multiplying large integers by divide-and-conquer

Question

Is there a smarter multiplication algorithm taking less than $O(n^2)$ many steps?

Answer

Remarkably, there is!

Table of Contents

History

- In 1952, one of the most famous mathematicians of the 20^{th} century, Andrey Kolmogorov, conjectured that you cannot multiply in less than quadratic many elementary operations.

- In 1960, Anatoly Karatsuba, then a 23-year-old student, found an algorithm (later it was called "divide-and-conquer") that multiplies two n-digit numbers in $\Theta\left(n^{\log_2 3}\right) \approx \Theta(n^{1.58\cdots})$ elementary steps, thus disproving the conjecture!! Kolmogorov was shocked!

The Karatsuba trick

- Once again we split each of our two input numbers A and B into halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- Previously we saw that

$$AB = A_1B_1 2^n + (A_1B_0 + A_0B_1)2^{\frac{n}{2}} + A_0B_0,$$

but rearranging the bracketed expression gives

$$AB = A_1B_1 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)2^{\frac{n}{2}} + A_0B_0,$$

saving one multiplication at each round of the recursion!

The Karatsuba trick

function $\textsc{Mult}(A, B)$
    if $|A| = |B| = 1$ **then return** AB
    else
        $A_1 \leftarrow \text{MoreSignificantPart}(A)$;
        $A_0 \leftarrow \text{LessSignificantPart}(A)$;
        $B_1 \leftarrow \text{MoreSignificantPart}(B)$;
        $B_0 \leftarrow \text{LessSignificantPart}(B)$;
        $U \leftarrow A_1 + A_0$;
        $V \leftarrow B_1 + B_0$;
        $X \leftarrow \textsc{Mult}(A_0, B_0)$;
        $W \leftarrow \textsc{Mult}(A_1, B_1)$;
        $Y \leftarrow \textsc{Mult}(U, V)$;
        return $W\,2^n + (Y - X - W)\,2^{n/2} + X$
    end if
end function

The Karatsuba trick

- How fast is this algorithm?

- Addition takes linear time, so we are only concerned with the number of multiplications.

- We need A_1B_1, A_0B_0 and $(A_1 + A_0)(B_1 + B_0)$; thus

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n.$$

The Karatsuba trick

Clearly, the run time $T(n)$ satisfies the recurrence

$$T(n) = 3\left[T\left(\frac{n}{2}\right)\right] + c\,n \tag{6}$$

Now the critical exponent is $c^* = \log_2 3$. Once again, we are in Case 1 of the Master Theorem, but this time

$$\begin{aligned} T(n) &= \Theta\left(n^{\log_2 3}\right) \\ &= \Theta\left(n^{1.58\ldots}\right) \\ &= o(n^2), \end{aligned}$$

disproving Kolmogorov's conjecture.

The Karatsuba trick

- We can do even better, by splitting the input numbers A and B into more than two pieces.
- In fact, for any $\epsilon > 0$ we can achieve $T(n) = O\left(n^{1+\epsilon}\right)$.
- However, with numbers divided into $p + 1$ pieces, the constant factors involved are on the order of p^p, so the algorithm is in practice hopelessly slow.
- Moral: in practice, asymptotic estimates are useless unless the size of the constants hidden by the big-oh notation are known to be reasonably small!

Table of Contents

Table of Contents

Quicksort revisited

Problem

Let A be an array of n distinct, comparable elements.

Sort A.

Note

We will assume that all array elements are distinct. This simplifies some of the later reasoning, but the same ideas hold when duplicates are allowed.

Quicksort

Algorithm

1. Designate the first element x as the *pivot*.
2. Partition A with all values smaller than x, followed by x at index j, followed by all values larger than x.
3. Recurse on $A[1..(j-1)]$ and on $A[(j+1)..n]$.

Note

We will take for granted that the second step (rearranging and partitioning the array) takes $\Theta(n)$. This is detailed in Section 7.1 of CLRS.

Complexity analysis of quicksort

- In the best case, the pivot is always the median, so the subarrays each have size about $n/2$. The recurrence is $T(n) = 2T(n/2) + \Theta(n)$, so case 2 of the Master Theorem gives

$$T(n) = \Theta(n \log n)$$

- In the worst case, the pivot is always the minimum (or maximum), so one subarray is empty and the other has size $n-1$. The recurrence is

$$T(n) = T(n-1) + \Theta(n).$$

While the Master Theorem does not apply, we can easily prove that $T(n) = \Theta(n^2)$.

Complexity analysis of quicksort

- However, the worst case is very rare. If the split is 9 : 1 at every step, there are only $\log_{10/9} n = O(\log n)$ many levels, each requiring a total of $O(n)$ work to partition. The overall time complexity in this case is still $\Theta(n \log n)$.

- The average case time complexity is $\Theta(n \log n)$. See Section 7.4 of CLRS for the proof.

Complexity analysis of quicksort

- The worst case of the stated algorithm includes sorted arrays and reverse sorted arrays, both of which are common use cases.
- To obfuscate the worst case, one can instead use a randomly chosen pivot, or the median of the first, last and middle element.
- However, the worst case performance is still $\Theta(n^2)$ using any of these strategies.

Note

Since partitioning takes linear time, a pivot selection strategy that also takes $O(n)$ time doesn't worsen the overall time complexity.

A related problem

Selection problem

Let A be an array of n distinct, comparable elements, and $1 \leq k \leq n$.

Find the kth smallest entry of A.

Attempt 1

Sort A in $O(n \log n)$ using merge sort, and return $A[k]$.

Question

Can we do better?

Quickselect

Algorithm

1. Designate the first element x as the *pivot*.
2. Partition A with all values smaller than x, followed by x at index j, followed by all values larger than x.
3. The recursive step depends on j.
   (a) If $j = k$, return x.
   (b) If $j > k$, recursively find the kth smallest element of $A[1..(j-1)]$.
   (c) If $j < k$, recursively find the $(k-j)$th smallest element of $A[(j+1)..n]$.

Complexity analysis of quickselect

- Again, the best case is when we always select the median as the pivot. The recurrence here is

$$T(n) = T(n/2) + \Theta(n),$$

and by case 3 of the Master Theorem, $T(n) = \Theta(n)$.

- The worst case is still $\Theta(n^2)$, when we always select an extreme value as the pivot.

- Again, any split of constant proportion (e.g. 9 to 1) is sufficient to achieve $\Theta(n)$ running time.
- It can be proven that the average case complexity is $\Theta(n)$: see Section 9.2 of CLRS.

Median

- The *median* is the "middle" value.

- Finding the median is a special case of the selection problem.
  - If n is odd, we find the $(n+1)/2$th smallest element.
  - There are multiple conventions for the median of an array of even length. We follow CLRS by finding the $n/2$th smallest element, the "lower median".

Finding the median

As before, we can find the median in:

- worst case $O(n \log n)$, using merge sort, or
- expected $O(n)$, using quickselect.

Question

Can we do better? Can we find the median (and indeed the kth smallest element) in worst case linear time?

Answer

Yes, using divide-and-conquer!

Table of Contents

Median-finding

Attempt 1

1. Divide A into $n/3$ blocks of three consecutive numbers.
2. From each block, discard the smallest number and largest number, keeping only the median.
3. Recursively find the median of the remaining $n/3$ numbers, the so-called "median of medians", and return it.

Median-finding

Question

Is this algorithm correct? In other words, is the median of medians the same as the median of the whole array?

Answer

No! The original is not even guaranteed to be the median of its block.

Consider for example

$$A = [1, 3, 5, \qquad 7, 9, 2, \qquad 4, 6, 8].$$

By only keeping the median of each block, we might discard the actual median prematurely. Can we trim the search space less aggressively?

Median of medians

Observations

- Suppose the median of the $n/3$ block medians is x.

- Half of the $n/3$ blocks have median less than x. In those blocks, the smallest number is even less, so it too is less than x.

- In the other half of the blocks, the median is greater than x, so the largest number is also greater than x.

Median of medians

In the following diagram, each column represents a sorted block, e.g. from an array [84, 53, 63, 79, . . . , 43]

53	45	21	8	11	19	27	3	43
63	79	40	60	33	37	76	48	62
84	83	70	88	98	51	82	96	69

For ease of visualisation, we will display the columns in sorted order of the middle row (the block medians).

11	19	21	3	8	43	53	27	45
33	37	40	48	60	62	63	76	79
98	51	70	96	88	69	84	82	83

Median of medians

11	19	21	3	8	43	53	27	45
33	37	40	48	60	62	63	76	79
98	51	70	96	88	69	84	82	83

- The element marked in yellow is the median of medians (although it is not the overall median).
- The smaller block medians are marked in cyan.
- In these blocks, the smallest entry is also smaller than the median of medians.
- The larger block medians are marked in magenta.
- In these blocks, the largest entry is also larger than the median of medians.

Median of medians

Conclusion

Ignoring rounding, there are $\frac{n}{3}$ blocks. Of these, $\frac{n}{6}$ have ‘small’ medians (cyan) and the other $\frac{n}{6}$ have ‘large’ medians (magenta).

The median of medians (yellow) might not be the median of the entire array, but it is definitely:

- greater than two of the three elements in $\frac{n}{6}$ blocks, and
- less than two of the three elements in $\frac{n}{6}$ blocks.

Therefore it is greater than $2 \times \frac{n}{6} = \frac{n}{3}$ array elements and less than the same number of array elements.

So it is in the middle third of the array by value.

Median of medians

Question

We can find a value in the middle third of the array. How does this help us find the median?

Answer

Recall that quickselect has good performance as long as the pivot is not too extreme!

We will use the median of medians *as a pivot selection strategy* in the full selection problem, and treat median-finding as merely a special case.

Quickselect using median of medians

Attempt 2

1. Divide A into $n/3$ blocks of three consecutive numbers.
2. Find the median of each block.
3. Recursively find the median of medians x, and designate it as the pivot.

4. Partition A with all values smaller than x, followed by x at index j, followed by all values larger than x.

5. The recursive step depends on j.
   (a) If $j = k$, return x.
   (b) If $j > k$, recursively find the kth smallest element of $A[1..(j-1)]$.
   (c) If $j < k$, recursively find the $(k-j)$th smallest element of $A[(j+1)..n]$.

Quickselect using median of medians

Complexity

1. Forming the blocks takes $\Theta(n)$ time.

2. Each block has only three items, so sorting to pick the median takes constant time per block.

3. Next, we recurse on the medians of the blocks, itself a selection problem that can be solved in $T(n/3)$ time.

4. Partitioning takes $\Theta(n)$ time.

5. The median of medians is in the middle third of the partitioned array, so the last step requires us to recursively perform selection on a subarray of at most $2n/3$ elements, taking $T(2n/3)$ time.

Quickselect using median of medians

Complexity (continued)

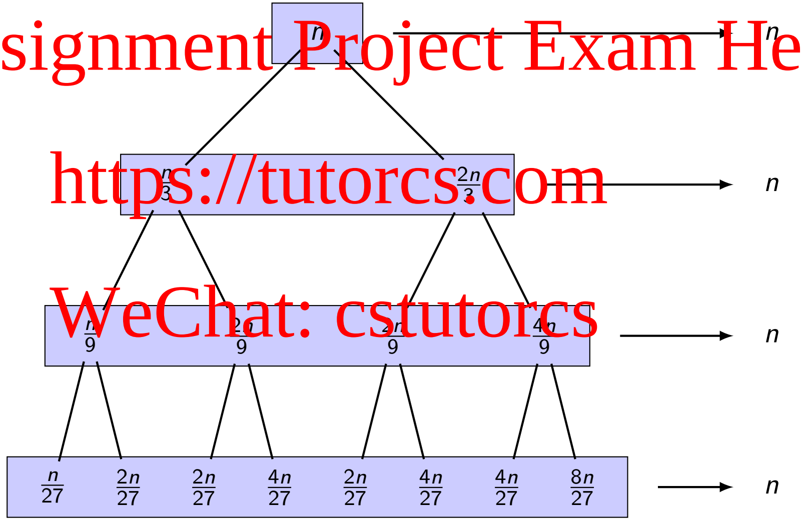
Therefore we can form the worst case recurrence

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + \Theta(n).$$

Unfortunately, this is not good enough. There are $\log_{3/2} n$ levels in this recursion, and again each requires a total of $\Theta(n)$ work, for an overall time complexiy of $\Theta(n \log n)$ in the worst case.

This is asymptotically no better than just sorting! Can we do better?

Quickselect using median of medians



Quickselect using median of medians: block size

- One parameter of our algorithm design seemed to be an arbitrary choice: blocks of size 3.
- What happens if we increase the block size to 5?
- There will be $\frac{n}{5}$ blocks, so step 3 will recurse on $\frac{n}{5}$ medians.
- The median of medians will be larger than the median (and hence also larger than the two smallest elements) of $\frac{n}{10}$ blocks, and smaller than the median (and hence also larger than the two largest elements) of the other $\frac{n}{10}$ blocks. Therefore the partition we achieve is between $3 : 7$ and $7 : 3$.

Quickselect using median of medians: blocks of five

Again, each column represents a sorted block, in increasing order of their median elements.

15	7	1	22	17	6	3	5	19
26	21	23	28	32	57	49	70	59
27	35	36	44	46	58	60	80	82
86	69	68	63	62	64	73	81	90
92	77	91	94	87	95	89	93	96

The element marked in yellow is the median of medians (although it is not the overall median). Those marked in cyan are guaranteed to be smaller than it, and those in magenta are guaranteed to be larger.

Quickselect using median of medians: blocks of five

Algorithm

1. Divide A into $n/5$ blocks of five consecutive numbers.
2. Find the median of each block.
3. Recursively find the median of medians x.

4. Partition A with all values smaller than x, followed by x at index j, followed by all values larger than x.
5. The recursive step depends on j.

   (a) If $j = k$, return x.
   (b) If $j > k$, recursively find the kth smallest element of $A[1..(j-1)]$.
   (c) If $j < k$, recursively find the $(k-j)$th smallest element of $A[(j+1)..n]$.

Quickselect using median of medians: blocks of five

Complexity

1. Forming the blocks takes $\Theta(n)$ time.

2. Each block has only five items, so sorting to pick the median takes constant time per block.

3. Next, we recurse on the medians of the blocks, itself a selection problem that can be solved in $T(n/5)$ time.

4. Partitioning takes $\Theta(n)$ time.

5. The median of medians is between the third and seventh deciles of A, so we recursively select from a subarray of at most $7n/10$ elements, taking $T(7n/10)$ time.

Quickselect using median of medians: blocks of five

Complexity (continued)

Therefore we can form the worst case recurrence

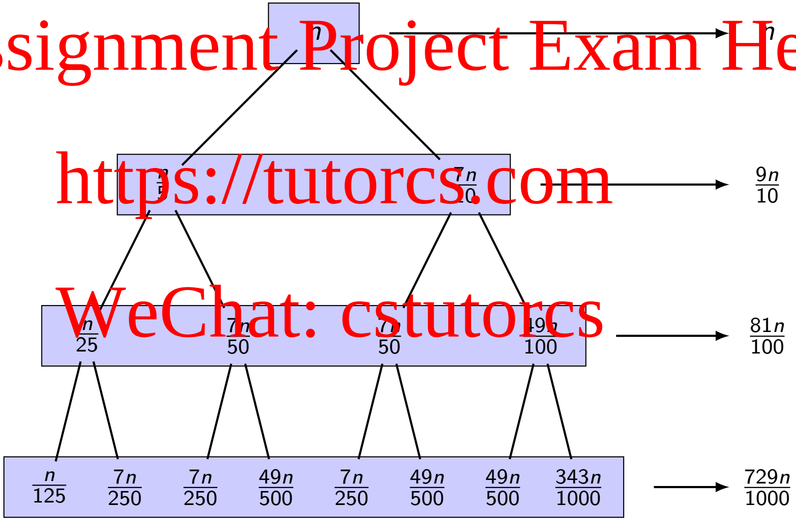
$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \Theta(n).$$

Surprisingly, this subtle change is enough to reduce the worst case time complexity from $\Theta(n \log n)$ to $\Theta(n)$.

The crucial difference is that

$$\frac{1}{3} + \frac{2}{3} = 1 > \frac{1}{5} + \frac{7}{10}.$$

Quickselect using median of medians: blocks of five



Quickselect using median of medians: blocks of five

Complexity (continued)

- Each subproblem requires us to form and sort blocks and then partition a subarray, taking time linear in the number of elements.
- We can therefore group multiple subproblems, which together will take time linear in their total number elements.
- The subarrays considered in the kth level have a total of $\left(\frac{9}{10}\right)^{k-1} n$ elements.
- Adding up over all levels, we get a geometric series with sum at most $10n$.
- Therefore we have succeeded in solving the selection problem in worst case linear time!

Quicksort using median of medians: blocks of five

The same pivot selection strategy allows us to develop a corresponding variant of quicksort.

Algorithm

1. Find the median x using median of medians quickselect, and designate it as the pivot.

2. Partition A with all values smaller than x, followed by x at index j, followed by all values larger than x.
3. Recurse on $A[1..(j-1)]$ and on $A[(j+1)..n]$.

Quicksort using median of medians: blocks of five

Complexity

- The first two steps each take linear time, and we are guaranteed to recurse into two subproblems each of size $n/2$.
- Therefore the recurrence is

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n),$$

and by case 2 of the Master Theorem, the worst case time complexity is $\Theta(n \log n)$.

Quicksort using median of medians: blocks of five

Question

Is this better than the original quicksort?

Answer

Yes and no.

- There is significant overhead involved in finding the median, so the constant factor hidden by the big-oh notation is substantial.
- In many practical applications this outweighs the improvement on the worst case.

Table of Contents

Puzzle

Problem

Five pirates have to split 100 bars of gold. They all line up and proceed as follows:

- The first pirate in line gets to propose a way to split up the gold (for example: everyone gets 20 bars)
- The pirates, including the one who proposed, vote on whether to accept the proposal. If the proposal is rejected, the pirate who made the proposal is killed.
- The next pirate in line then makes his proposal, and the 4 pirates vote again. If the vote is tied (2 vs 2) then the proposing pirate is still killed. Only majority can accept a proposal.

This process continues until a proposal is accepted or there is only one pirate left.

Puzzle

Problem (continued)

Assume that every pirate has the same priorities in the following order:

1. not having to walk the plank;
2. getting as much gold as possible;
3. seeing other pirates walk the plank, just for fun.

Puzzle

Problem (continued)

What proposal should the first pirate make?

Hint

Assume first that there are only two pirates, and see what happens. Then assume that there are three pirates and that they have figured out what happens if there were only two pirates and try to see what they would do. Further, assume that there are four pirates and that they have figured out what happens if there were only three pirates, try to see what they would do. Finally assume there are five pirates and that they have figured out what happens if there were only four pirates.

That's All, Folks!!