Introduction to the theory of computation
week 8 (Course Notes, chapters 3 & 5)

MergeSort complexity: loose ends

Last time we considered MergeSort:

```
MergeSort(A, f, l)
# Precondition:  0 ≤ f ≤ l ≤ length(A) − 1
# Postcondition:  A[f..l] has the same elements as before the invocation,
                  in sorted order; and all other elements of A are unchanged
if f = l then                       # already sorted
        return
else
        m := (f + l) div 2          # m is the middle index of A[f..l]
        MergeSort(A, f, m)          # sort the first half
        MergeSort(A, m + 1, l)      # sort the second half
        Merge(A, f, m, l)           # merge the two sorted halves
end if
```

... and decided that the number of steps it took was bounded by $T(n)$, defined by:

$$T(n) = \begin{cases} c, & n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + dn, & n > 1 \end{cases}.$$

Although the formula for $T(n)$ neglects some constant steps from each call to $T(n)$ (the overhead of the `if` statement, and time to calculate m), this doesn't affect $T(n)$'s validity as a lower bound. If you want to use $T(n)$ as an upper bound, increasing the parameter d more than accounts for these constant steps.

There are a few issues we have to deal with before coming up with a closed-form bound on $T(n)$.

Issue 1: Does the given definition work? Since the definition is given recursively, it is enough to verify that $1 \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$ when $n > 1$, to see that $T(n)$ is well-defined (a short proof by induction will do this). Lemma 3.5 from the Course Notes show us that:

Claim: For any integer $n > 1$, you have $1 \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$.

Proof: For odd n, $\lceil n/2 \rceil = (n+1)/2$, and for even n, $\lceil n/2 \rceil = n/2 < (n+1)/2$, so in either case $\lceil n/2 \rceil \leq (n+1)/2$. By assumption, $n > 1$, so $2n > n + 1$, and $n > (n+1)/2 \geq \lceil n/2 \rceil$. Thus, $\lceil n/2 \rceil < n$ if $n > 1$. By definition, $\lfloor n/2 \rfloor \leq \lceil n/2 \rceil$, and when $n > 1$, you have $n \geq 2$, so $n/2 \geq 1$, so $\lfloor n/2 \rfloor \geq 1$. Putting these inequalities together gives you

$$1 \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n \qquad \text{for } n > 1.$$

This is what was to be proved. QED.

ISSUE 2: Does the algorithm break n roughly in half (or at least into strictly smaller pieces)? In other words, if $n = l - f + 1$ (number of array elements in $A[f..l]$, and if $l - f > 0$ (so we set $m = \lfloor (f+l)/2 \rfloor$), then is $m - f + 1 = \lceil n/2 \rceil$ and is $l - (m+1) + 1 = \lfloor n/2 \rfloor$? If I define $\epsilon = (f+l)/2 - \lfloor (f+l)/2 \rfloor$, then $1 > \epsilon \geq 0$, and

$$\begin{aligned} l - (m+1) + 1 &= l - m = l - \left\lfloor \frac{f+l}{2} \right\rfloor = l - \frac{f+l}{2} + \epsilon = \left\lceil l - \frac{f+l}{2} \right\rceil = \left\lceil \frac{l-f}{2} \right\rceil \\ &= \left\lfloor \frac{l-f+1}{2} \right\rfloor = \left\lfloor \frac{n}{2} \right\rfloor \end{aligned}$$

(Verify $\lceil k/2 \rceil = \lfloor (k+1)/2 \rfloor$ for $k \in \mathbb{N}$, see Course Notes, Section 5.8, Theorem 5.14) This exercise shows that if `MergeSort` is called on an array of size $n > 1$, then it recursively calls itself on sub-arrays of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Recall from Lecture 1 that $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$.

ISSUE 3: Can we solve a special case of the recurrence? There doesn't seem to be a consistent way to evaluate $T(\lceil n/2 \rceil)$ or $T(\lfloor n/2 \rfloor)$, so you can make the simplifying assumption that $n = 2^k$ for some $k \in \mathbb{N}$ — n is a natural power of 2. Now you can re-write $T(n)$ as

$$T(2^k) = \begin{cases} c, & k = 0 \\ 2T(2^{k-1}) + d2^k, & k > 0 \end{cases}.$$

Now there is a recursive pattern that can be revealed by "unwinding" $T(2^k)$:

$$\begin{aligned} T(2^k) &= 2T(2^{k-1}) + d2^k \\ &= 2(2T(2^{k-2}) + d2^{k-1}) + d2^k = 2^2 T(2^{k-2}) + 2d2^k \\ &= 2^2(2T(2^{k-3}) + d2^{k-2}) + 2d2^k = 2^3 T(2^{k-3}) + 3d2^k \\ &\vdots \\ &= 2^k T(2^{k-k}) + kd2^k = 2^k c + kd2^k = 2^k(c + kd). \end{aligned}$$

Notice how this expression works even when $k = 0$. However, the vertical dots amount to (careful) hand-waving, so we'll need to use something (induction) to make the reasoning precise. You can use PSI:

CLAIM: $P(k)$: "$T(2^k) = 2^k(c + kd)$" is true for all $k \in \mathbb{N}$.

PROOF (INDUCTION ON n): $P(0)$ states that $T(1) = c$, which is certainly true, given the definition of $T(n)$.

INDUCTION STEP: I want to show that $P(k) \Rightarrow P(k+1)$, so I assume that $P(k)$ is true for some arbitrary $k \in \mathbb{N}$. Now I unwind $T(2^{k+1})$ a little, and then use the inductive hypothesis (IH) on it:

$$T(2^{k+1}) = 2T(2^k) + 2^{k+1}d = 2(2^k(c + kd)) + 2^{k+1}d = 2^{k+1}(c + [k+1]d)$$

This is exactly what $P(k+1)$ asserts, so $P(k) \Rightarrow P(k+1)$.

I conclude that $P(k)$ is true for all $k \in \mathbb{N}$. QED.

Translating this result back into $n = 2^k$, this says that whenever n is a natural power of 2, then $T(n) = nc + n \log_2(n) d$.

ISSUE 4: What about all the other n that aren't powers of 2? We'd like to show that if $n = 2^x$ (x is a non-negative real number), then $T(2^x) \leq T(2^{\lceil x \rceil})$. This would be immediate if we knew that $T(n)$ were monotonic (so $m < n$ would imply that $T(m) \leq T(n)$), so that's the next thing to show.

CLAIM: $P(n)$ "For every positive integer m less than n, $T(m) \leq T(n)$" is true for all $n \in \mathbb{N} - \{0\}$.

PROOF (COMPLETE INDUCTION ON n): You need to show that for an arbitrary positive integer n, $P(\{1, \ldots, n-1\}) \Rightarrow P(n)$. So, for an arbitrary $n \in \mathbb{N} - \{0\}$, assume $P(\{1, \ldots, n-1\})$.

$P(1)$ is trivially true, because there are no positive integers m less than 1. $P(2)$ states that $T(1) \leq T(2)$, or in other words $c \leq 2T(1) + 2d = 2c + 2d$, which is also true because c and d are not negative.

Now consider $P(n)$ when $n > 2$. This means that $1 < n - 1 < n$, and (by Lemma 3.5, above) $1 \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil \leq n$. Since you've assume $P(\{1, \ldots, n-1\})$, in particular you have assumed $P(n-1)$, $P(\lfloor n/2 \rfloor)$, and $P(\lceil n/2 \rceil)$. By assuming $P(n-1)$ you've assumed that $T(n-1)$ is no smaller than $T(m)$ for any positive natural number m less than $n-1$, so all you need now is to show that $T(n-1) \leq T(n)$. We have (by definition, since $n - 1 > 1$):

$$\begin{aligned} T(n-1) &= T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + d(n-1) \\ [\text{by } P(\lfloor n/2 \rfloor), P(\lceil n/2 \rceil), \text{ and } d \geq 0] \quad &\leq T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + dn \\ &= T(n). \end{aligned}$$

Therefore, $T(n-1) \leq T(n)$, so $P(\{1, \ldots, n-1\}) \Rightarrow P(n)$, as wanted.

Conclude that $P(n)$ is true for all $n \in \mathbb{N} - \{0\}$, in other words, if m and n are natural numbers with $m < n$, then $T(m) \leq T(n)$.

BOUNDING $T(n)$

Suppose n is an arbitrary natural number (not necessarily a natural power of 2). You know that n is no bigger than the next largest natural power of 2 $\widehat{n}$, that is

$$n/2 < 2^{\lfloor \log_2 n \rfloor} \leq n = 2^{\log_2 n} \leq 2^{\lceil \log_2 n \rceil} = \widehat{n} < 2n.$$

Now you can use what you already know about $T(2^{\lceil \log_2 n \rceil})$ to create an upper bound for $T(n)$, let $2^{\lceil \log_2 n \rceil} = \widehat{n}$ for convenience, and for any $n \geq 2$:

$$\begin{aligned} [T(n) \text{ monotone, and } \widehat{n} \text{ is a power of 2}] \qquad T(n) &\leq T(\widehat{n}) = \widehat{n}(c + d \log_2 \widehat{n}) \\ [\log \text{ monotone, and } \widehat{n} < 2n] &\leq 2n(c + d \log_2[2n]) = 2n(c + d + d \log_2 n) \\ [\log_2 n \geq 1 \text{ if } n \geq 2] &\leq 2n(c \log_2 n + d \log_2 n + d \log_2 n) \\ [\text{where } \kappa = (2c + 4d)] &= \kappa n \log_2 n \end{aligned}$$

In other words, $T(n)$ is big-oh of $n \log n$. A very similar computation shows that $T(n) \geq \kappa' n \log_2 n$ ($\kappa' \neq \kappa$), which means that $T(n)$ is big-omega of $n \log n$. Let $2^{\lfloor \log_2 n \rfloor} = \check{n}$, and since $T(n)$ is monotone, for $n \geq 4$:

$$\begin{aligned} T(n) &\geq T(\check{n}) = \check{n}(c + d \log_2 \check{n}) \\ [n/2 < \check{n}, \text{ and } \log_2 \text{ monotone}] &\geq \frac{n}{2}(c + d \log_2(n/2)) = \frac{n}{2}(c + d \log_2 n - d) \\ [\log_2 n \geq 2 \text{ for } n \geq 4] &= \frac{n}{2}\left(c + \frac{d}{2} \log_2 n + d\left[\frac{\log_2 n}{2} - 1\right]\right) \\ [c, d \text{ are nonnegative. Let } \kappa' = d/4.] &\geq \frac{n}{2}\left(\frac{d}{2} \log_2 n\right) = \kappa' n \log_2 n. \end{aligned}$$

The meaning of this is that if you drew the graph of $T(n)$, you could sandwich it between $\kappa \log n$, and $\kappa' \log n$ (except for the first few values, perhaps).

Divide and conquer: the Master Theorem (Chapter 3)

The strategy of `MergeSort` can be generalized to other algorithms. The idea is to divide the input of size n into pieces that are of size roughly n/b (for some positive natural number b), solve the smaller problems, and combine them using a technique that is polynomial in n, that is requiring dn^l, for some natural number l. Assume there is some ad-hoc method for solving very small instances of the problem (between 1 and b), and these take no more than c steps. Since n/b can hardly be expected to be a natural number for every n, there will be a_1 pieces of size $\lceil n/b \rceil$, and a_2 pieces of size $\lfloor n/b \rfloor$. Putting all this together gives you a recursive relation for $T(n)$:

$$T(n) = \begin{cases} c, & 1 \leq n < b \\ a_1 T\left(\left\lceil \frac{n}{b} \right\rceil\right) + a_2 \left(\left\lfloor \frac{n}{b} \right\rfloor\right) + dn^l, & n \geq b \end{cases}$$

You can get a closed form for $T(n)$ following the same steps as for `MergeSort`:

1. Solve the problem when n is a power of b, that is $n = b^k$. In Assignment 2, you will show that, if $a = a_1 + a_2$, then

$$T(n) = a^k T(1) + dn^l \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^l}\right)^j = cn^{\log_b a} + dn^l \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^l}\right)^j.$$

   This is still not in closed form, due to the sum at the end. This is a geometric series with $\log_b(n)$ terms. Using a trick attributed to Gauss, and setting $a/b^l = \alpha$ for convenience, this becomes

$$\sum_{j=0}^{\log_b(n)-1} \alpha^j = \begin{cases} (1 - \alpha^{\log_b(n)})/(1-\alpha), & \alpha \neq 1 \\ \log_b(n), & \alpha = 1 \end{cases}.$$

   This means that $T(n)$ will have different forms according to whether α is less than, equal to, or greater than, one. In the Course Notes pages 90 and 91 you can work out the algebra for these three cases to show that if n is a natural power of b, then for some non-negative constant κ:

$$T(n) \leq \begin{cases} \kappa n^l, & a < b^l \\ \kappa n^l \log_b(n), & a = b^l \\ \kappa n^{\log_b(a)}, & a > b^l \end{cases}.$$

2. Prove that $T(n)$ is monotone increasing (similar to the proof for `MergeSort`).

3. Extend the bound on $T(n)$ to all n, not just powers of b. Derive a closed-form upper bound for $T(n)$, for some non-negative constant λ:

$$T(n) \leq \begin{cases} \lambda n^l, & a < b^l \\ \lambda n^l \log_b(n), & a = b^l \\ \lambda n^{\log_b(a)}, & a > b^l \end{cases}.$$

4. Derive a lower bound on $T(n)$ for some non-negative constant λ':

$$T(n) \geq \begin{cases} \lambda' n^l, & a < b^l \\ \lambda' n^l \log_b(n), & a = b^l \\ \lambda' n^{\log_b(a)}, & a > b^l \end{cases}.$$

This gives an upper and lower bound for an entire class of algorithms.

Propositional formulas and truth (Chapter 5)

From CSC165 you know that propositions (aka statements) are sentences that can be evaluated to exactly one value from $\{\text{true}, \text{false}\}$. For example:

True: "Water is wet."

False: "Snow is carnivorous."

You can combine old propositions to get new ones using connectives:

Conjunction: "Water is wet and snow is carnivorous." (false, because and requires both propositions to be true).

Disjunction: "Water is wet or snow is carnivorous." (true, because or requires at least one proposition to be true).

Conditional: "Water is wet implies that snow is carnivorous." (false, since we can have wet water without carnivorous snow).

Biconditional: "Water is wet if and only if snow is carnivorous." (false again).

Negation: "It is not true that snow is carnivorous." (true, because it negates something false).

The English language is a verbose and slippery object. Long and complex propositions become very difficult to accurately evaluate. The notation of propositional formulas allows us to make complex propositional formulas briefer and (usually) clearer.

Propositional variables: Our set of propositional variables, PV contains strings representing the simplest (most primitive) propositions we are currently considering.

Propositional formulas: Our set of propositional formulas, F_{PV}, is the smallest set such that

1. Any propositional variable in PV belongs to F_{PV}.
2. F_{PV} is closed under the following operations on members P_1 and P_2:
   (a) $f_1(P_1) = \neg P_1$.
   (b) $f_2(P_1, P_2) = (P_1 \rightarrow P_2)$.
   (c) $f_3(P_1, P_2) = (P_1 \leftrightarrow P_2)$.
   (d) $f_4(P_1, P_2) = (P_1 \wedge P_2)$.
   (e) $f_5(P_1, P_2) = (P_1 \vee P_2)$.

Operators $f_2, \ldots, f_4$ insert one of the binary connectives $\{\rightarrow, \leftrightarrow, \wedge, \vee\}$ between its arguments and wrap the result with parentheses. Operator f_1 prepends the unary connective $\neg$ to its argument and leaves out the parentheses.

By applying the basis and induction step you can build an arbitrarily complex formula with an unambiguous structure — each separate formula can be uniquely parsed by matching left and right parentheses (see the Unique Readability Theorem in the Course Notes). This means that there is one and only one way to read the formula as one or more sub-formulas combined with a connective. Here's a propositional formula:

$$((\neg x \wedge (y \vee z)) \rightarrow (\neg((x \wedge z) \vee y) \rightarrow (x \vee z)))$$

Unique readability comes at the cost of very thick layers of parentheses for even moderately complex formulas. There are agreed-upon conventions that allow us to (informally)write formulas with fewer parentheses by using rules of precedence (some operators bind "tighter" than others) and associativity:

1. Omit the outer parentheses.
2. $\wedge$ and $\vee$ have precedence over $\rightarrow$ and $\leftrightarrow$, while $\wedge$ has precedence over $\vee$.
3. If the same connective is used several times in a row, grouping is assumed to be to the right. Using these rules, re-write the example above:

$$\begin{aligned} ((\neg x \wedge (y \vee z)) &\rightarrow (\neg((x \wedge z) \vee y) \rightarrow (x \vee z))) \\ \neg x \wedge (y \vee z) &\rightarrow \neg(x \wedge z \vee y) \rightarrow x \vee z \end{aligned}$$

So far we have defined what is a well-formed (legal) propositional formula. However, these are meaningless strings unless you assign truth values to the basis elements (propositional variables), and have consistent rules for truth values of more complicated variables. Assigning truth values can be thought of as setting the current state of the universe (what's true, what's false).