Part I

Algorithm Design And Analysis

Chapter 1

Basic Techniques

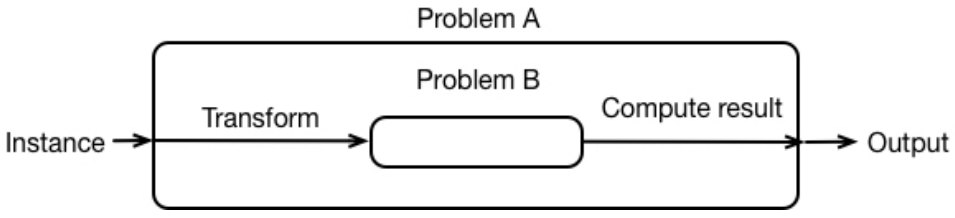
This chapter covers two basic algorithm-design techniques: reduction and brute force.

1 Algorithmic Reduction

Definition 1.1 (Algorithmic Reduction)**.** We say that a problem A is ***reducable*** to another problem B, if any instance of A can be reduced to one or more instances of B. This means that it is possible to solve any instance of A by following the a three step process (also illustrated below).

- Transform the instance of problem A to one or many instances of problem B.
- Solve all instances of problem B.
- Use results to B instances to compute result to the A instance.

We sometimes refer to problem B as a ***subproblem*** of A, especially if the reduction involves producing multiple instances of B.



Efficiency of Reduction. The efficiency of an algorithm obtained via reduction depends on the complexity of the problem being reduced to, and the cost of transforming the instances and putting together the results to compute final result. It is therefore important

to ensure that the costs of the instance transformation and the cost of computing the final result remain small in comparison to the cost of the problem being reduced to.

We generally think of a reduction as being ***efficient*** if the total cost of the input and output transformations are asymptotically the same as that of the problem being reduced to.

Example 1.1 (Reduction from Maximal to Sorting)**.** We can reduce the problem of finding the maximal element in a sequence to the problem of (comparison) sorting. To this end, we sort the sequence in ascending order and then select the last element, which is one the largest elements in the input sequence.

In this reduction, we don't need to transform the input. To compute the final result, we retrieve the last element in the sequence, which usually requires logarithmic work or less. Because comparison sorting itself requires $\Theta(n \lg n)$ work for a sequence of n elements, the reduction is efficient. But, the resulting algorithm is not a work-efficient: it requires $\Theta(n \lg n)$ work, whereas we can find the maximum element of a sequence in $\Theta(n)$ work.

Example 1.2 (Reduction from Minimal to Maximal)**.** Suppose that we wish to find the minimal element in a sequence but only know how to find the maximal element efficiently in linear work. We can reduce the minimality problem to the maximality problem by inverting the sign of all the elements in the input, finding the maximal value of the transformed input, and then inverting the sign of the result to obtain the minimum. Because these computations require linear work in the size of the input sequence, and because finding the maximal element requires linear work, this reduction is quite efficient.

The resulting algorithm, which requires linear work in total, is also asymptotically work efficient.

Remark. Reduction is a powerful technique and can allow us to solve a problem by reducing it to another problem, which might appear very different or even unrelated. For example, we might be able to reduce a problem on strings such as that of finding the shortest superstring of a set of strings to a graph problem. We have seen an example of this earlier in genome sequencing.

Remark (Proving Hardness by Reduction). Reduction technique can also be used "in the other direction" to show that some problem is at least as hard as another or to establish a lower bound. In particular, if we know (or conjecture) that problem A is hard (e.g., requires exponential work), and we can reduce it to problem B (e.g., using polynomial work), then we know that B must also be hard. Such reductions are central to the study of complexity theory.

2 Brute Force

The brute-force technique involves enumerating all possible solutions, a.k.a., ***candidate solutions*** to a problem, and checking whether each candidate solution is valid. Depending on the problem, a solution may be returned as soon as it is found, or the search may continue until a better solution is found.

Example 1.3 (Brute-Force Sorting)**.** We are asked to sort a set of keys a. As a first algorithm, we can apply the brute force technique by considering the candidate-solution space (sequence of keys), enumerating this space by considering each candidate, and checking that it is sorted.

More precisely speaking, this amounts to trying all permutations of a and testing that each permutation is sorted. We return as soon as we find a sorted permutation. Because there are $n!$ permutations and $n!$ is very large even for small n, this algorithm is inefficient. For example, using Stirling's approximation, we know that $100! \approx \frac{100}{e}^{100} \geq 10^{100}$. There are only about 10^{80} atoms in the universe so there is probably no feasible way we could apply the brute force method directly. We conclude that this approach to sorting is not ***tractable***, because it would require a lot of energy and time for all but tiny inputs.

Remark. The total number of atoms in the (known) universe, which is less than 10^{100} is a good number to remember. If your solution requires this much work or time, it is probably intractable.

Important. Brute-force algorithms are usually naturally parallel. Enumerating all candidate solutions (e.g., all permutations, all elements in a sequence) is typically easy to do in parallel and so is testing that a candidate solution is indeed a valid solution. But brute-force algorithms are usually not work efficient and therefore not desirable—in algorithm design, our first priority is to minimize the total work and only then the span.

Example 1.4 (Maximal Element)**.** Suppose that we are given a sequences of natural numbers and we wish to find the maximal number in the sequence. To apply the brute force technique, we first identify the solution space, which consists of the elements of the input sequence. The brute-force technique advises uses to try all candidate solutions. We thus pick each element and test that it is greater or equal to all the other elements in the sequence. When we find one such element, we know that we have found the maximal element.

This algorithm requires $\Theta(n^2)$ comparisons and can be effective for very small inputs, but it is far from an optimal algorithm, which would perform $\Theta(n)$ comparisons. .

Example 1.5 (Brute-Force Overlaps)**.** Given two strings a and b, let's define the (maximum) ***overlap*** between a and b as the largest suffix of a that is also a prefix of b. For example, the overlap between "15210" and "2105" is "210", and the overlap between "15210" and "1021" is "10".

We can find the overlap between a and b by using the brute force technique. We first note that the solution space consists of the suffixes of a that are also prefixes of b. To apply brute force, we consider each possible suffix of a and check if it is a prefix of b. We then select the longest such suffix of a that is also a prefix of b.

The work of this algorithm is $|a| \cdot |b|$, i.e., the product of the lengths of the strings. Because we can try all positions in parallel, the span is determined by the span of checking that a particular prefix is also a suffix, which is $O(\lg |b|)$. Selecting the maximum requires $O(\lg |a|)$ span. Thus the total span is $O(\lg (|a| + |b|))$.

Exercise 1.1. The analysis given in the example above is not "tight" in the sense that there is a bound that is asymptotically dominated by $O(|a| \cdot |b|)$. Can you improve on the bound?

Example 1.6 (Brute-Force Shortest Paths)**.** Consider the following problem: we are given a graph where each edge is assigned distance (e.g., a nonnegative real number), and asked to find the shortest path between two given vertices in the graph.

Using brute-force, we can solve this problem by enumerating the solution space, which is the set of all paths between the given two vertices, and selecting the path with the smallest total distance. To compute the total distance over the path, we sum up the distances of all edges on the path.

Example 1.7 (Brute-Force Shortest Distances)**.** Consider the following variant of the shortest path problem: we are given a graph where each edge is assigned a distance (e.g., a nonnegative real number), and asked to find the shortest distance between two given vertices in the graph. This problem differs from the one above, because it asks for the shortest distance rather than the path.

Using brute-force, we might want to solve this problem by enumerating the solution set, which is the real numbers starting from zero, and selecting the smallest one. But, this is impossible, because real numbers are not countable, and cannot be enumerated.

One solution is to use the reduction technique and reduce the problem to the shortest-path version of the problem described above and then compute the distance of the returned path.

As an additional refinement, we can reformulate the shortest-path problem to return the distance of the shortest path in addition to the path itself. With this refinement, computing the final result requires no additional computation beyond returning the distance.

Important (Strengthening)*.* We used an important technique in the shortest-paths example where we refined the problem that we are reducing to so that it returns us more information than strictly necessary. This technique is called ***strengthening*** and is commonly employed to improve efficiency by reducing redundancy.

Example 1.8 (Brute-Force Shortest Paths (Decision Version))**.** Consider the "decision problem" variant of the shortest path problem: we are given a graph where each edge is assigned a distance (e.g., a nonnegative real number), and we are given a ***budget***, which is a distance value. We are asked to check whether there is a path between two given vertices in the graph that is shorter than the given budget. This problem differs from the shortest path problem, because it asks us to return a "Yes" or "No" answer. Such problems are called ***decision problems***.

Using brute force, we could enumerate all candidate solutions, which are either "Yes" or "No", and test whether each one is indeed a valid solution. In this case, this does not help, because we are back to our original problem that we started with. Again, we can "refine" our brute-force approach a bit by reducing it to the original shortest-path problem , and then checking whether the distance of the resulting path is larger than the budget.

Remark. Experienced algorithms designers perform reductions between different variants of the problem as shown in the last two examples "quietly" (Example **??** and Example **??**). That is, they don't explicitly state the reduction but simply apply the brute-force technique to a slightly different set of candidate solutions. For beginners and even for experienced designers, however, recognizing such reductions can be instructive.

Remark (Utility of Brute Force). Even though brute-force algorithms are usually inefficient, they can be useful.

- Brute-force algorithms are usually easy to design and are a good starting point toward a more efficient algorithm. They can help in understanding the structure of the problem, and lead to a more efficient algorithm.
- In practice, a brute-force algorithm can help in testing the correctness of a more efficient algorithm by offering a solution that is easy to implement correctly.

Chapter 2

Divide and Conquer

This chapter describes the divide-and-conquer technique, an important algorithm-design technique, and applies it to several problems. Divide and conquer is an effective technique for solving a variety of problems, and usually leads to efficient and parallel algorithms.

1 Divide and Conquer

A divide-and-conquer algorithm has a distinctive anatomy: it has a base case to handle small instances and an inductive step with three distinct phases: "divide", "recur", and "combine." The ***divide*** phase divides the problem instance into smaller instances; the ***recur*** phase solves the smaller instances; and the ***combine*** phase combines the results for the smaller instance to construct the result to the larger instance.

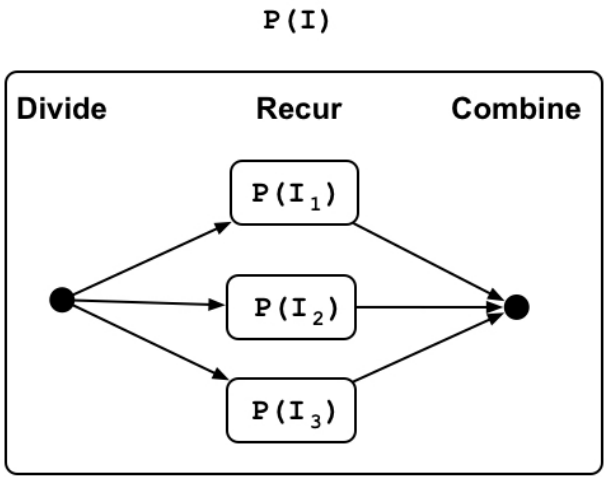
Definition 2.1 (Divide-And-Conquer Algorithm)**.** A divide-and-conquer algorithm has the following structure.

Base Case: When the instance I of the problem P is sufficiently small, compute the solution $P(I)$ perhaps by using a different algorithm.

Inductive Step:

1. **Divide** instance I into some number of smaller instances of the same problem P.
2. **Recur** on each of the smaller instances and compute their solutions.
3. **Combine** the solutions to obtain the solution to the original instance I.

Example 2.1. The drawing below illustrates the structure of a divide-and-conquer algorithm that divides the problem instance into three independent subinstances.



Properties of Divide-and-Conquer Algorithms. Divide-and-Conquer has several important properties.

- It follows the structure of an inductive proof, and therefore usually leads to relatively simple proofs of correctness. To prove a divide-and-conquer algorithm correct, we first prove that the base case is correct. Then, we assume by strong (or structural) induction that the recursive solutions are correct, and show that, given correct solutions to smaller instances, the combined solution is correct.

- Divide-and-conquer algorithms can be work efficient. To ensure efficiency, we need to make sure that the divide and combine steps are efficient, and that they do not create too many sub-instances.

- The work and span for a divide-and-conquer algorithm can be expressed as a mathematical equation called recurrence , which can be usually be solved without too much difficulty.

- Divide-and-conquer algorithms are naturally parallel, because the sub-instances can be solved in parallel. This can lead to significant amount of parallelism, because each inductive step can create more independent instances. For example, even if the algorithm divides the problem instance into two subinstances, each of those subinstances could themselves generate two more subinstances, leading to a geometric progression, which can quickly produce abundant parallelism.

Analysis of Divide-and-Conquer Algorithms. Consider an algorithm that divides a problem instance of size n into $k > 1$ independent subinstances of sizes $n_1, n_2, \ldots n_k$, recursively solves the instances, and combine the solutions to construct the solution to the original instance.

We can write the work of such an algorithm using the recurrence then

$$W(n) \;=\; W_{\text{divide}}(n) \;+\; \sum_{i=1}^{k} W(n_i) \;+\; W_{\text{combine}}(n) + 1.$$

The work recurrence simply adds up the work across all phases of the algorithm (divide, recur, and combine).

To analyze the span, note that after the instance is divided into subinstance, the subinstances can be solved in parallel (because they are independent), and the results can be combined. The span can thus be written as the recurrence:

$$S(n) = S_{\text{divide}}(n) \;+\; \max_{i=1}^{k} S(n_i) \;+\; S_{\text{combine}}(n) + 1.$$

Note. The work and span recurrences for a divide-and-conquer algorithm usually follow the recursive structure of the algorithm, but is a function of size of the arguments instead of the actual values.

Example 2.2 (Maximal Element)**.** We can find the maximal element in a sequence using divide and conquer as follows. If the sequence has only one element, we return that element, otherwise, we divide the sequence into two equal halves and recursively and in parallel compute the maximal element in each half. We then return the maximal of the results from the two recursive calls. For a sequence of length n, we can write the work and span for this algorithm as recurrences as follows:

$$W(n) = \begin{cases} \Theta(1) & \text{if} \quad n \leq 1 \\ 2W(n/2) + \Theta(1) & \text{otherwise} \end{cases}$$

$$S(n) = \begin{cases} \Theta(1) & \text{if} \quad n \leq 1 \\ S(n/2) + \Theta(1) & \text{otherwise.} \end{cases}$$

This recurrences yield

$$W(n) = \Theta(n) \text{ and}$$
$$S(n) = \Theta(\lg n).$$

Algorithm 2.2 (Reduce with Divide and Conquer)**.** The ***reduce*** primitive performs a computation that involves applying an associative binary operation op to the elements of a sequence to obtain (reduce the sequence to) a final value. For example, reducing the sequence $\langle\, 0, 1, 2, 3, 4 \,\rangle$ with the $+$ operation gives us $0 + 1 + 2 + 3 + 4 = 10$. If the operation requires constant work (and thus span), then the work and span of a reduction is $\Theta(n)$ and $\Theta(\lg n)$ respectively.

We can write the code for the reduce primitive on sequences as follows.

```
reduceDC f id a =
  if isEmpty a then
    id
  else if isSingleton a then
    a[0]
  else
    let
      (l, r) = splitMid a
      (a, b) = (reduceDC f id l || reduceDC f id r)
    in
      f(a, b)
end
```

2 Merge Sort

In this section, we consider the comparison sorting problem and the merge-sort algorithm, which offers a divide-and-conquer algorithm for it.

Definition 2.3 (The Comparison-Sorting Problem)**.** Given a sequence a of elements from a universe U, with a total ordering given by $<$, return the same elements in a sequence r in sorted order, i.e. $r[i] \leq r[i+1], 0 < i \leq |a| - 1$.

Algorithm 2.4 (Merge Sort)**.** Given an input sequence, merge sort divides it into two sequences that are approximately half the length, sorts them recursively, and merges the sorted sequences. Mergesort can be written as follows.

```
mergeSort a =
  if |a| ≤ 1 then
    a
  else
    let
      (l, r) = splitMid a
      (l', r') = (mergeSort l || mergeSort r)
    in
      merge(l', r')
    end
```

Note. In the merge sort algorithm given above the base case is when the sequence is empty or contains a single element. In practice, however, instead of using a single element or empty sequence as the base case, some implementations use a larger base case consisting of perhaps ten to twenty keys.

Correctness and Cost. To prove correctness we first note that the base case is correct. Then by induction we note that l' and r' are sorted versions of l and r. Because l and r together contain exactly the same elements as a, we conclude that `merge` (l', r') returns a sorted version of a.

For the work and span analysis, we assume that merging can be done in $\Theta(n)$ work and $\Theta(\lg n)$ span, where n is the sum of the lengths of the two sequences. We can thus write the work and span for this sorting algorithm as

$$W(n) = \begin{cases} \Theta(1) & \text{if} \quad n \leq 1 \\ 2W(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

$$S(n) = \begin{cases} \Theta(1) & \text{if} \quad n \leq 1 \\ S(n/2) + \Theta(\lg n) & \text{otherwise.} \end{cases}$$

The recurrences solve to

$$W(n) = \Theta(n \lg n)$$
$$S(n) = \Theta(\lg^2 n).$$

Remark (Quick Sort). Another divide-and-conquer algorithm for sorting is the quick-sort algorithm. Like merge sort, quick sort requires $\Theta(n \log n)$ work, which is optimal for the comparison sorting problem, but only "in expectation" over random decisions that it makes during its execution. While merge sort has a trivial divide step and interesting combine step, quick sort has an interesting divide step but trivial combine step. We will study quick sort in greater detail.

3 Sequence Scan

Intuition for Scan with Divide and Conquer. To develop some intuition on how to design a divide-and-conquer algorithm for the sequence scan problem, let's start by dividing the sequence in two halves, solving each half, and then putting the results together.

For example, consider the sequence $\langle\, 2, 1, 3, 2, 2, 5, 4, 1 \,\rangle$. If we divide in the middle and scan over the two resulting sequences we obtain (b, b') and (c, c'), such that

$$(b, b') = (\langle\, 0, 2, 3, 6 \,\rangle, 8), \text{ and}$$
$$(c, c') = (\langle\, 0, 2, 7, 11 \,\rangle, 12).$$

Note now that b already gives us the first half of the solution. To compute the second half, observe that in calculating c in the second half, we started with the identity instead of the sum of the first half, b'. Therefore, if we add the sum of the first half, b', to each element of c, we would obtain the desired result.

Algorithm 2.5 (Scan with Divide and Conquer)**.** By refining the intuitive description above, we can obtain a divide-and-conquer algorithm for sequences scan, which is given below.

```
scanDC f id a =
  if |a| = 0 then
    (⟨ ⟩, id)
  else if |a| = 1 then
    (⟨ id ⟩, a[0])
  else
    let
      (b, c) = splitMid a
      ((l, b'), (r, c')) = (scanDC f id b || scanDC f id c)
      r' = ⟨ f(b', x) : x ∈ r ⟩
    in
      (append (l, r'), f(b', c'))
    end
```

Remark. Observe that this algorithm takes advantage of the fact that id is really the identity for f, i.e. $f(id, x) = x$.

Cost Analysis. We consider the work and span for the algorithm. Note that the combine step requires a map to add b' to each element of c, and then an append. Both these take $O(n)$ work and $O(1)$ span, where $n = |a|$. This leads to the following recurrences for the whole algorithm:

$$\begin{array}{lcll} W(n) & = & 2W(n/2) + O(n) & \in \ O(n \log n) \\ S(n) & = & S(n/2) + O(1) & \in \ O(\log n). \end{array}$$

Although this is much better than $O(n^2)$ work, we can do better by using another design technique called contraction.

4 Euclidean Traveling Salesperson Problem

We consider a variant of the well-known Traveling Salesperson Problem (TSP) and design a divide-and-conquer heuristic for it. This variant, known as the Euclidean Traveling Salesperson Problem (eTSP), is **NP** hard. It requires solving the TSP problem in graphs where the vertices (e.g., cities) lie in a Euclidean space and the edge weights (e.g., distance measure between cities) is the Euclidean distance. More specifically, we're interested in the planar version of the eTSP problem, defined as follows:

Definition 2.6 (The Planar Euclidean Traveling Salesperson Problem)**.** Given a set of points P in the 2-d plane, the ***planar Euclidean traveling salesperson*** (eTSP) problem is to find a tour of minimum total distance that visits all points in P exactly once, where the distance between points is the Euclidean (i.e. ℓ_2) distance.

Example 2.3. Assuming that we could go from one place to another using your personal airplane, this is the problem we would want to solve to find a minimum length route visiting your favorite places in Pittsburgh.

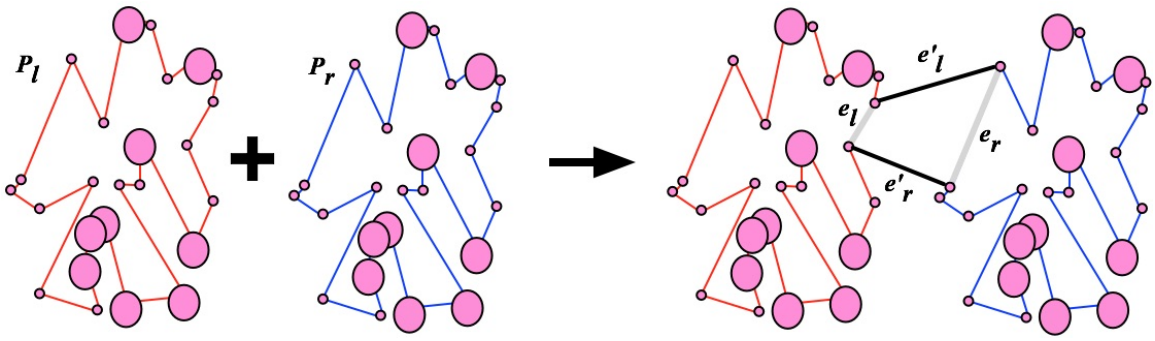
As with the TSP, eTSP is **NP**-hard, but it is easier to approximate. Unlike the TSP problem, which only has constant approximations, it is known how to approximate this problem to an arbitrary but fixed constant accuracy ε in polynomial time (the exponent of n has $1/\varepsilon$ dependency). That is, such an algorithm is capable of producing a solution that has length at most $(1+\varepsilon)$ times the length of the best tour.

Note. In Section **??**, we cover another approximation algorithm for a metric variant of TSP that is based on Minimum Spanning Trees (MST). That approximation algorithm gives a constant-approximation guarantee.

Intuition for a Divide and Conquer Algorithm for eTSP. We can solve an instance of the eTPS problem by splitting the points by a cut in the plane, solving the eTSP instances on the two parts, and then merging the solutions in some way to construct a solution for the original problem.

For the cut, we can pick a cut that is orthogonal to the coordinate lines. We could for example find the dimension along which the points have a larger spread, and then cut just below the median point along that dimension.

This division operation gives us two smaller instances of eTSP, which can then be solved independently in parallel, yielding two cycles. To construct the solution for the original problem, we can merge the solutions. To merge the solution in the best possible way, we can take an edge from each of the two smaller instances, remove them, and then bridge the end points across the cut with two new edges. For each such pair of edges, there are two possible ways that we can bridge them, because when we are on the one side, we can jump to any one of the endpoints of the two bridges. To construct, the best solution, we can try out which one of these yields the best solution and take that one.



To choose which swap to make, we consider all pairs of edges of the recursive solutions consisting of one edge $e_\ell = (u_\ell, v_\ell)$ from the left and one edge $e_r = (u_r, v_r)$ from the right and determine which pair minimizes the increase in the following cost:

$$swapCost((u_\ell, v_\ell), (u_r, v_r)) = \|u_\ell - v_r\| + \|u_r - v_\ell\| - \|u_\ell - v_\ell\| - \|u_r - v_r\|$$

where $\|u - v\|$ is the Euclidean distance between points u and v.

Algorithm 2.7 (Divide-and-Conquer eTSP)**.** By refining the intuition describe above, we arrive at a divide-and-conquer algorithm for solving eTPS, whose pseudo-code is shown below.

```
eTSP (P) =
  if |P| < 2 then
    raise TooSmall
  else if |P| = 2 then
    ⟨ (P[0], P[1]), (P[1], P[0]) ⟩
  else
    let
      (P_ℓ, P_r) = split P along the longest dimension
      (L, R) = (eTSP P_ℓ) || (eTSP P_r)
      (c, (e, e')) = minVal_first {(swapCost(e, e'), (e, e')) : e ∈ L, e' ∈ R}
    in
        swapEdges (append (L, R), e, e')
    end
```

The function $minVal_{first}$ uses the first value of the pairs to find the minimum, and returns the (first) pair with that minimum. The function $swapEdges(E, e, e')$ finds the edges e and e' in E and swaps the endpoints. As there are two ways to swap, it picks the cheaper one.

Remark. This heuristic divide-and-conquer algorithm is known to work well in practice.

Cost Analysis. Let's analyze the cost of this algorithm in terms of work and span. We have

$$W(n) = 2W(n/2) + O(n^2)$$
$$S(n) = S(n/2) + O(\log n)$$

We have already seen the recurrence $S(n) = S(n/2)+O(\log n)$, which solves to $O(\log^2 n)$. Here we'll focus on solving the work recurrence.

To solve the recurrence, we apply a theorem proven earlier , and obtain

$$W(n) = O(n^2).$$

Strengthening. In applications of divide-and-conquer technique that we have consider so far in this chapter, we divide a problem instance into instances of the same problem. For example, in sorting, we divide the original instance into smaller instances of the sorting problem. Sometimes, it is not possible to apply this approach to solve a given problem, because solving the same problem on smaller instances does not provide enough information to solve the original problem. Instead, we will need to gather more information when

solving the smaller instances to solve the original problem. In this case, we can ***strengthen*** the original problem by requiring information in addition to the information required by the original problem. For example, we might strengthen the sorting problem to return to us not just the sorted sequence but also a histogram of all the elements that count the number of occurrences for each element.

5 Divide and Conquer with Reduce

Many divide-and-conquer algorithms have the following structure, where $emptyVal$, $base$, and $myCombine$ span for algorithm specific values.

```
myDC a =
  if |a| = 0 then
    emptyVal
    else if |a| = 1 then
      base(a[0])
    else
      let (l, r) = splitMid a in
        (l', r') = (myDC l || myDC r)
      in
        myCombine (l', r')
      end
```

Algorithms that fit this pattern can be implemented in one line using the sequence $reduce$ function. Turning a divide-and-conquer algorithm into a reduce-based solution is as simple as invoking $reduce$ with the following parameters

$$reduce\ myCombine\ emptyVal\ (map\ base\ a).$$

Important. This pattern does not work in general for divide-and-conquer algorithms. In particular, it does not work for algorithms that do more than an simple split that partitions their input in two parts in the middle. For example, it cannot be used for implementing the quick-sort algorithm, because the divide step partitions the data with respect to a pivot. This step requires picking a pivot, and then filtering the data into elements less than, equal, and greater than the pivot. It also does not work for divide-and-conquer algorithms that split more than two ways, or make more than two recursive calls.

Chapter 3

Contraction

This chapter describes the contraction technique for algorithm design and applies it to several problems.

1 Contraction Technique

A contraction algorithm has a distinctive anatomy: it has a base case to handle small instances and an inductive step with three distinct phases: "contract, "recur", and "expand." It involves solving recursively smaller instances of the same problem and then expanding the solution for the larger instance.

Definition 3.1 (Contraction)**.** A contraction algorithm for problem P has the following structure.

Base Case: If the problem instance is sufficiently small, then compute and return the solution, possibly using another algorithm.

Inductive Step(s): If the problem instance is sufficiently large, then

- Apply the following two steps, as many times as needed.
  1. ***Contract:*** "contract", i.e., map the instance of the problem P to a smaller instance of P.
  2. ***Solve:*** solve the smaller instance recursively.
- ***Expand*** the solutions to smaller instance to solve the original instance.

Remark. Contraction differs from divide and conquer in that it allows there to be only one *independent* smaller instance to be recursively solved. There could be multiple *dependent* smaller instances to be solved one after another (sequentially).

Properties of Contraction. Contraction algorithms have several important properties.

- Due to their inductive structure, we can establish the correctness of a contraction algorithm using principles of induction: we first prove correctness for the base case, and then prove the general (inductive) case by using strong induction, which allows us to assume that the recursive call is correct.
- The work and span of a contraction algorithm can be expressed as a mathematical recurrence that reflects the structure of the algorithm itself. Such recurrences can then unually be solved using well-understood techniques, and without significant difficulty.
- Contraction algorithms can be work efficient, if they can reduce the problem size geometrically (by a constant factor greater than 1) at each contraction step, and if the contraction and the expansions steps are efficient.
- Contraction algorithms can have a low span (high parallelism), if size of the problem instance decreases geometrically, and if contraction and expansion steps have low spans.

Example 3.1 (Maximal Element)**.** We can find the maximal element in a sequence a using contraction as follows. If the sequence has only one element, we return that element, otherwise, we can map the sequence a into a sequence b which is half the length by comparing the elements of a at consecutive even-odd positions and writing the larger into b. We then find the largest in b and return this as the result.

For example, we map the sequence $\langle\, 1, 2, 4, 3, 6, 5 \,\rangle$ to $\langle\, 2, 4, 6 \,\rangle$. The largest element of this sequence, 6 is then the largest element in the input sequence.

For a sequence of length n, we can write the work and span for this algorithm as recurrences as follows

$$W(n) = \begin{cases} \Theta(1) & \text{if} \quad n \leq 1 \\ W(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

$$S(n) = \begin{cases} \Theta(1) & \text{if} \quad n \leq 1 \\ S(n/2) + \Theta(1) & \text{otherwise.} \end{cases}$$

Using the techniques discussed at the end of this chapter, we can solve the recurrences to obtain $W(n) = \Theta(n)$ and $S(n) = \Theta(\lg n)$.

2 Reduce with Contraction

The ***reduce*** primitive performs a computation that involves applying an associative binary operation op to the elements of a sequence to obtain (reduce the sequence to) a final value. For example, reducing the sequence $\langle\, 0, 1, 2, 3, 4 \,\rangle$ with the $+$ operation gives us $0 + 1 + 2 + 3 + 4 = 10$. Recall that the type signature for $reduce$ is as follows.

$$reduce\ (f : \alpha * \alpha \to \alpha)\ (id : \alpha)\ (a : \mathbb{S}_\alpha) : \alpha,$$

where f is a binary function, a is the sequence, and id is the left identity for f.

Even though we can define *reduce* broadly for both associative and non-associative functions, in this section, we assume that the function f is associative.

Generalizing the algorithm for computing the maximal element leads us to an implementation of an important parallelism primitive called *reduce*. The crux in using the contraction technique is to design an algorithm for reducing an instance of the problem to a geometrically smaller instance by performing a parallel contraction step. To see how this can be done, consider instead applying the function f to consecutive pairs of the input.

For example if we wish to compute the sum of the input sequence

$$\langle\, 2, 1, 3, 2, 2, 5, 4, 1 \,\rangle$$

by using the addition function, we can contract the sequence to

$$\langle\, 3, 5, 7, 5 \,\rangle\,.$$

Note that the contraction step can be performed in parallel, because each pair can be considered independently in parallel.

By using this contraction step, we have reduced the input size by a factor of two. We next solve the resulting problem by invoking the same algorithm and apply expansion to construct the final result. We note now that by solving the smaller problem, we obtain a solution to the original problem, because the sum of the sequence remains the same as that of the original. Thus, the expansion step requires no additional work.

Algorithm 3.2 (Reduce with Contraction)**.** An algorithm for *reduce* using contraction is shown below; for simplicity, we assume that the input size is a power of two.

```
(* Assumption: |a| is a power of 2 *)
reduceContract f id a =
  if |a| = 1 then
    a[0]
  else
    let
      b = ⟨ f(a[2i], a[2i + 1]) : 0 ≤ i < ⌊|a|/2⌋ ⟩
    in
      reduceContract f id b
    end
```

Cost of Reduce with Contraction. Assuming that the function being reduced over performs constant work, parallel tabulate in the contraction step requires linear work, we can thus write the work of this algorithm as

$$W(n) = W(n/2) + n.$$

This recurrence solves to $O(n)$.

Assuming that the function being reduced over performs constant span, parallel tabulate in the contraction step requires constant span; we can thus write the work of this

algorithm as

$$S(n) = S(n/2) + 1.$$

This recurrence solves to $O(\log n)$.

3 Scan with Contraction

We describe how to implement the $scan$ sequence primitive efficiently by using contraction. Recall that the $scan$ function has the type signature

$$scan\ (f : \alpha * \alpha \to \alpha)\ (id : \alpha)\ (a : \mathbb{S}_\alpha)\ :\ (\mathbb{S}_\alpha * \alpha)$$

where f is an associative function, a is a sequence, and id is the identity element of f. When evaluated with a function and a sequence, $scan$ can be viewed as applying a reduction to every prefix of the sequence and returning the results of such reductions as a sequence.

Example 3.2. Applying $scan$ ' $+$ ', i.e., "plus scan" on the sequence $\langle\, 2, 1, 3, 2, 2, 5, 4, 1 \,\rangle$ returns

$$(\langle\, 0, 2, 3, 6, 8, 10, 15, 19 \,\rangle, 20)\,.$$

We will use this as a running example.

Based on its specification, a direct algorithm for $scan$ is to apply a reduce to all prefixes of the input sequence. Unfortunately, this easily requires quadratic work in the size of the input sequence.

We can see that this algorithm is inefficient by noting that it performs lots of redundant computations. In fact, two consecutive prefixes overlap significantly but the algorithm does not take advantage of such overlaps at all, computing the result for each overlap independently.

By taking advantage of the fact that any two consecutive prefixes differ by just one element, it is not difficult to give a linear work algorithm (modulo the cost of the application of the argument function) by using iteration.

Such an algorithm may be expressed as follows

$$scan\ f\ id\ a = h\,(iterate\ g\ (\langle\,\rangle, id)\ a)\,,$$

where

$$g((b, y), x) = ((append\ \langle\, y \,\rangle\ b),\ f(y, x))$$

and

$$h(b, y) = ((reverse\ b), y)$$

where $reverse$ reverses a sequence.

This algorithm is correct but it almost entirely sequential, leaving no room for parallelism.

Scan via Contraction, the Intuition. Because *scan* has to compute some value for each prefix of the given sequence, it may appear to be inherently sequential. We might be inclined to believe that any efficient algorithms will have to keep a cumulative "sum," computing each output value by relying on the "sum" of the all values before it.

We will now see that we can implement *scan* efficiently using contraction. To this end, we need to reduce a given problem instance to a geometrically smaller instance by applying a contraction step. As a starting point, let's apply the same idea as we used for implementing reduce with contraction .

Applying the contraction step from the *reduce* algorithm described above, we would reduce the input sequence

$$\langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$$

to the sequence

$$\langle 3, 5, 7, 5 \rangle,$$

which if recursively used as input would give us the result

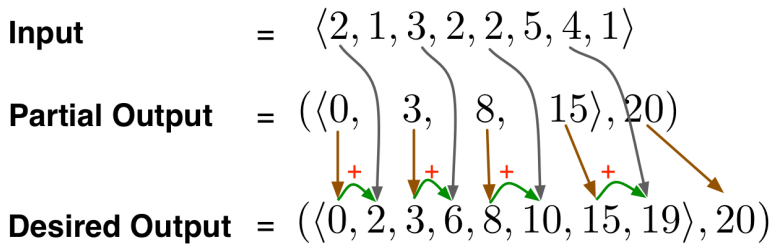
$$(\langle 0, 3, 8, 15 \rangle, 20).$$

Notice that in this sequence, the elements in even numbered positions are consistent with the desired result:

$$(\langle 0, 2, 3, 6, 8, 10, 15, 19 \rangle, 20).$$

Half of the elements are correct because the contraction step, which pairs up the elements and reduces them, does not affect, by associativity of the function being used, the result at a position that do not fall in between a pair.

To compute the missing element of the result, we will use an expansion step and compute each missing elements by applying the function element-wise to the corresponding elements in the input and the results of the recursive call to *scan*. The drawing below illustrates this expansion step.



Algorithm 3.3 (Scan Using Contraction, for Powers of 2)**.** Based on the intuitive description above, we can write the pseudo-code for *scan* as follows. For simplicity, we assume that n is a power of two.

```
(* Assumption: |a| is a power of two.  *)
scan f id a =
  if |a| = 0 then
  (⟨ ⟩, id)
  else if |a| = 1 then
    (⟨ id ⟩, a[0])
  else
    let
      a' = ⟨ f(a[2i], a[2i + 1]) : 0 ≤ i < n/2 ⟩
      (r, t) = scan f id a'
    in
      (⟨ p_i : 0 ≤ i < n ⟩, t), where p_i = { r[i/2]              even(i)
                                            { f(r[i/2], a[i − 1])  otherwise
    end
```

Cost of Scan with Contraction. Let's assume for simplicity that the function being applied has constant work and constant span. We can write out the work and span for the algorithm as a recursive relation as

$$W(n) = W(n/2) + n, \text{ and}$$
$$S(n) = S(n/2) + 1,$$

because 1) the contraction step which tabulates the smaller instance of the problem performs linear work in constant span, and 2) the expansion step that constructs the output by tabulating based on the result of the recursive call also performs linear work in constant span.

These recursive relations should look familiar. They are the same as those that we ended up with when we analyzed the work and span of our contraction-based implementation of *reduce* and yield

$$W(n) = O(n)$$
$$S(n) = O(\log n).$$

Chapter 4

Maximum Contiguous Subsequence Sum

This chapter reviews the classic problem of finding the contiguous subsequence of a sequence with the maximal value, and provides several algorithms for the problem by applying several design techniques including brute force , reduction , and divide and conquer .

1 The Problem

Definition 4.1 (Subsequence)**.** A ***subsequence*** b of a sequence a is a sequence that can be derived from a by deleting zero or more elements of a without changing the order of remaining elements.

Example 4.1. Several examples follow.

- The sequence $\langle\, 0, 2, 4 \,\rangle$ is a subsequence of $\langle\, 0, 1, 2, 2, 3, 4, 5 \,\rangle$.
- The sequence $\langle\, 2, 4, 3 \,\rangle$ is a not subsequence of $\langle\, 0, 1, 2, 2, 3, 4, 5 \,\rangle$ but $\langle\, 2, 3, 4 \,\rangle$ is.

Definition 4.2 (Contiguous Subsequence)**.** A ***contiguous subsequence*** is a subsequence that appears contiguously in the original sequence. For any sequence a of n elements, the subsequence

$$b = a[i \cdots j], 0 \leq i \leq j < n,$$

consisting of the elements of a at positions $i, i+1, \ldots, j$ is a contiguous subsequence of b.

Example 4.2. For $a = \langle 1, -2, 0, 3, -1, 0, 2, -3 \,\rangle$, here are some contiguous subsequences:

- $\langle\, 1 \,\rangle$,

- $\langle\ -2, 0, 3\ \rangle$, and
- $\langle\ 3, -1, 0, 2, -3\ \rangle$.

The sequence $\langle\ -1, 2, -3\ \rangle$ is not a contiguous subsequence, even though it is a subsequence.

Definition 4.3 (The Maximum Contiguous Subsequence (MCS) Problem)**.** Given a sequence of integers, the ***Maximum Contiguous Subsequence Problem*** (MCS) requires finding the contiguous subsequence of the sequence with maximum total sum, i.e.,

$$\mathsf{MCS}\,(a) = \arg\max 0 \leq i, j < |a| \left(\sum_{k=i}^{j} a[k] \right).$$

We define the sum of an empty sequence to be $-\infty$.

Example 4.3. For $a = \langle\, 1, -2, 0, 3, -1, 0, 2, -3\,\rangle$, a maximum contiguous subsequence is, $\langle\, 3, -1, 0, 2\,\rangle$; another is $\langle\ 0, 3, -1, 0, 2\ \rangle$.

Definition 4.4 (The Maximum Contiguous Subsequence Sum (MCSS) Problem)**.** Given a sequence of integers, the ***Maximum Contiguous Subsequence Sum Problem*** (MCSS)

requires finding the total sum of the elements in the contiguous subsequence of the sequence with maximum total sum, i.e.,

$$\mathsf{MCSS}\,(a) = \max \left\{ \sum_{k=i}^{j} a[k]\ :\ 0 \leq i, j < |a| \right\}.$$

Example 4.4. For $a = \langle\, 1, -2, 0, 3, -1, 0, 2, -3\ \rangle\rangle$, a maximum contiguous subsequence is, $\langle\, 3, -1, 0, 2\,\rangle$; another is $\langle\ 0, 3, -1, 0, 2\ \rangle$. Thus $\mathsf{MCSS}\ (a) = 4$.

For the empty sequence, $\mathsf{MCSS} = -\infty$ because the sum of an empty sequence is defined as $-\infty$.

Note. Here we only consider sequences of integers and the addition operation to compute the sum, the techniques that we describe should apply to sequences of other types and other associative sum operations.

Lower Bound. To solve the MCSS problem, we need to inspect, at the very least, each and every element of the sequence. This requires linear work in the length of the sequence and thus solve the MCSS problem requires $\Omega(n)$ work.

Note (History of the Problem). The study of maximum contiguous subsequence problem goes to 1970's. The problem was first proposed in by Ulf Grenander, a Swedish statistician and a professor of applied mathematics at Brown University, in 1977. The problem has several names, such maximum subarray sum problem, or maximum segment sum problem, the former of which appears to be the name originally used by Grenander. Grenander intended it to be a simplified model for maximum likelihood estimation of patterns in digitized images, whose structure he wanted to understand.

According to Jon Bentley (Jon Bentley, Programming Pearls (1st edition), page 76.) in 1977, Grenander described the problem to Michael Shamos of Carnegie Mellon University who overnight designed a divide and conquer algorithm, which corresponds to our first divide-and-conquer algorithm . When Shamos and Bentley discussed the problem and Shamos' solution, they thought that it was probably the best possible. A few days later Shamos described the problem and its history at a Carnegie Mellon seminar attended by statistician Joseph (Jay) Kadane, who designed the work efficient algorithm within a minute. Kadane's algorithm correspond to the linear work and span algorithm described below.

Roadmap. The remaining sections apply various algorithm-design techniques to the MCS and MCSS problems. To exercise our vocabulary for algorithm design, the content is organized to identify carefully the design techniques, sometimes at a level of precision that may, especially in subsequent reads, feel pedantic.

2 Brute Force

This section presents a first solution to the MCSS problem by using the brute force technique .

Algorithm 4.5 (MCSS: Brutest Force)**.** We can solve the MCSS problem by brute force. First, we identify the set of candidate solutions as the set of all integers. Then we enumerate all integers and, for each one, check that there is a contiguous subsequence whose sum is equal to that integer. We stop when we find the largest integer with a matching subsequence.

Perhaps obviously, such an algorithm would not terminate, because we don't know when to stop. Notice, however, that the solution is bounded by the sum of all positive integers in the sequence. We can thus stop the search when we reach that bound.

This algorithm terminates but has the undesirable characteristic that its bound depends on the values of the elements in the sequence rather that its length.

Reduction to MCS. Our first algorithm is rather inefficient in the worst case, because it tries a large number of candidate solutions. We can achieve a better bound by reducing MCSS problem to the Maximum Contiguous Subsequence (MCS) problem , which requires finding the contiguous subsequence with the largest sum.

The reduction itself is straightforward: because both problems operate on the same input, there is no need to transform the input. To compute the output, we sum the elements in the sequence returned by the MCS problem. Using $reduce$, this requires $O(n)$ work and $O(\lg n)$ span. Thus, the work and span of the reduction is $O(n)$ and $O(\lg n)$ respectively.

Algorithm 4.6 (MCS: Brute Force)**.** We can solve the MCS problem by brute force: we enumerate all candidate solutions, which consist of all the contiguous subsequences of the sequence, and find the one with the largest sum. To generate all contiguous subsequences,

we can generate all pairs of integers (i, j), $0 \leq i \leq j < n$, compute the sum of the subsequence that corresponds to the pair, and pick the one with the largest total. We write the algorithm as follows:

$MCSBF\ a =$
  let
    $maxSum\ ((i,j,s),(k,\ell,t)) =$ if $s > t$ then (i,j,s) else (k,ℓ,t)
    $b = \langle\ (i, j, reduce\ +\ 0\ a[i \cdots j]) : 0 \leq i \leq j < n\ \rangle$
    $(i, j, s) = reduce\ maxSum\ (-1,-1,-\infty)\ b$
  in
    (i, j)
  end

Cost of Brute Force MCS. Using array sequence costs, generating the n^2 subsequences requires a total of $O(n^2)$ work and $O(\lg n)$ span. Reducing over each subsequence using *reduce* adds linear work per subsequence, bringing the total work to $O(n^3)$. The final reduce that select the maximal subsequence require $O(n^2)$ work. The total work is thus dominated by the computation of sums for each subsequence, and therefore is $O(n^3)$.

Because we can generate all pairs in parallel and compute their sum in parallel in $\Theta(\lg n)$ span using *reduce*, the algorithm requires $\Theta(\lg n)$ span.

Algorithm 4.7 (MCSS: Brute Force)**.** Our first algorithm uses brute force technique and a reduction to the MCS problem, which we again solve by brute force using the brute-force MCS algorithm . We can write the algorithm as follows:

$MCSSBF\ a =$
  let
    $(i, j) = MCSBF\ a$
    $sum = reduce\ '+'\ 0\ a[i \cdots j]$
  in
    sum
  end.

Strengthening. The brute force algorithm has some redundancy: to find the solution, it computes the result for the MCS problem and then computes the sum of the result sequence, which is already computed by the MCS algorithm. We can eliminate this redundancy by strengthening the MCS problem and requiring it to return the sum in addition to the subsequence.

Exercise 4.1. Describe the changes to the algorithms Algorithm **??** and Algorithm **??** to implement the strengthening described above. How does strengthening impact the work and span costs?

Algorithm 4.8 (MCSS: Brute Force Strengthened)**.** We can write the algorithm based on strengthening directly as follows. Because the problem description requires returning only

the sum, we simplify the algorithm by not tracking the subsequences.

```
MCSSBF a =
  let
    b = ⟨ reduce + 0 a[i ··· j] : 0 ≤ i ≤ j < n ⟩
  in
    reduce max −∞ b
  end
```

Cost Analysis. Let's analyze the work and span of the strengthened brute-force algorithm by using array sequences and by appealing to our cost bounds for $reduce$, $subseq$, and $tabulate$. The cost bounds for enumerating all possible subsequences and computing their sums is as follows.

$$\begin{aligned} W(n) &= 1 + \sum_{1 \le i \le j \le n} W_{reduce}(j-i) \le 1 + n^2 \cdot W_{reduce}(n) &= \Theta(n^3) \\ S(n) &= 1 + \max_{1 \le i \le j \le n} S_{\text{reduce}}(j-i) \le 1 + S_{reduce}(n) &= \Theta(\lg n) \end{aligned}$$

The final step of the brute-force algorithm is to find the maximum over these $\Theta(n^2)$ combinations. Since the reduce for this step has $\Theta(n^2)$ work and $\Theta(\lg n)$ span the cost of the final step is subsumed by other costs analyzed above. Overall, we have an $\Theta(n^3)$-work $\Theta(\lg n)$-span algorithm.

Note. Note that the span requires the maximum over $\binom{n}{2} \le n^2$ values, but since $\lg n^k = k \lg n$, this is simply $\Theta(\lg n)$.

Summary. When trying to apply the brute-force technique to the MCSS problem, we encountered a problem. We solved this problem by reducing MCSS problem to another problem, MCS. We then realized a redundancy in the resulting algorithm and eliminated that redundancy by strengthening MCS. This is a quite common route when designing a good algorithm: we find ourselves refining the problem and the solution until it is (close to) perfect.

3 Applying Reduction

In the previous section , we used the brute-force technique to develop an algorithm that has logarithmic span but large (cubic) work. In this section, we apply the reduction technique to obtain a low span and work-efficient (linear work) algorithm for the MCSS problem.

3.1 Auxiliary Problems

Overlapping Subsequences and Redundancy. To understand how we might improve the amount of work, observe that the brute-force algorithm performs a lot of repeated and

thus redundant work. To see why, consider the subsequences that start at some location. For each position, e.g., the middle, the algorithm considers a subsequence that starts at the position and at any other position that comes after it. Even though each subsequence differs from another by a single element (in the ending positions), the algorithm computes the total sum for each of these subsequences independently, requiring linear work per subsequence. The algorithm does not take advantage of the overlap between the many subsequences it considers.

Reducing Redundancy. We can reduce redundancy by taking advantage of the overlaps and computing all subsequences that start or end at a given position together. Our basic strategy in applying the reduction technique is to use this observation. To this end, we define two problems that are closely related to the MCSS problem, present efficient and parallel algorithm for these problems, and then reduce the MCSS to them.

Definition 4.9 (MCSSS)**.** The ***Maximum Contiguous Subsequence Sum with Start***, abbreviated ***MCSSS***, problem requires finding the maximum contiguous subsequence of a sequence that starts at a given position.

Definition 4.10 (MCSSE Problem)**.** The ***Maximum Contiguous Subsequence with Ending***, i.e., the ***MCSSE*** problem requires finding the maximum contiguous subsequence ending at a specified end position.

Reducing MCSS to MCSSS and MCSSE. Observe that we can reduce the MCSS problem to MCSSS problem by enumerating over all starting positions, solving MCSSS for each position, and taking the maximum over the results. A similar reduction works for the MCSSE problem.

Because the inputs to all these problems are essentially the same, we don't need to transform the input. To compute the output for MCSS, we need to perform a $reduce$. The reduction itself is thus efficient.

Algorithm 4.11 (An Optimal Alggorithm for MCSSS)**.** We can solve the MCSSS problem by first computing the sum for all subsequences that start at the given position using $scan$ and then finding their maximum.

$$
\begin{array}{l}
MCSSSOpt\ a\ i = \\
\quad \texttt{let} \\
\quad\quad b = scanI\ '+'\ 0\ a\ [i \cdots (|a| - 1)] \\
\quad \texttt{in} \\
\quad\quad reduce\ max\ {-\infty}\ b \\
\quad \texttt{end}
\end{array}
$$

Because the algorithm performs one scan and one reduce, it performs $\Theta(n - i)$ work in $\Theta(\lg n - i)$ span. This is asymptotically optimal because, any algorithm for the MCSSS problem must inspect as least $n - i$ elements of the input sequence.

Algorithm 4.12 (An Optimal Algorithm for MCSSE)**.** To solve the MCSSE problem efficiently and in low span, we observe that any contiguous subsequence of a given sequence

can be expressed in terms of the difference between two prefixes of the sequence: the subsequence $A[i \cdots j]$ is equivalent to the difference between the subsequence $A[0 \cdots j]$ and the subsequence $A[0 \cdots i-1]$.

Thus, we can compute the sum of the elements in a contiguous subsequence as

$$\mathit{reduce}\ '+'\ 0\ a[i \cdots j] = (\mathit{reduce}\ '+'\ 0\ a[0 \cdots j]) - (\mathit{reduce}\ '+'\ 0\ a[0 \cdots i-1])$$

where the "-" is the subtraction operation on integers.

This observation leads us to a solution to the MCSSE problem. Consider an ending position j and suppose that we have the sum for each prefix that ends at $i < j$. Since we can express any subsequence ending at position j by subtracting the corresponding prefix, we can compute the sum for the subsequence $A[i \cdots j]$ by subtracting the sum for the prefix ending at j from the prefix ending at $i-1$. Thus the maximum contiguous sequence ending at position j starts at position i which has the minimum of all prefixes up to i. We can compute the minimum prefix that comes before j by using just another scan. These observations lead to the following algorithm.

```
MCSSEOpt a j =
  let
    (b, v) = scan '+' 0 a[0 ··· j]
    w = reduce min ∞ b
  in
    v − w
  end
```

Using array sequences, this algorithm performs $\Theta(j)$ work and $\Theta(\lg(j))$ span. This is optimal because any algorithm for MCSSE must inspect at least j elements of the input sequence.

3.2 Reduction to MCSSS

Algorithm 4.13 (MCSS: Reduced Force)**.** We can find a more efficient brute-force algorithm for MCSS by reducing the problem to MCSSS and using the optimal algorithm for it .

The idea is to try all possible start positions, solve the MCSSS problem for each, and select their maximum. The code for the algorithm is shown below.

```
MCSSReducedForce a =
  reduce max −∞ ⟨ (MCSSSOpt a i) : 0 ≤ i < n ⟩ .
```

In the worst case, the algorithm performs $\Theta(n^2)$ work in $\Theta(\lg n)$ span, delivering a linear-factor improvement in work.

Remark. By reducing MCSS to MCSSS, we were able to eliminate a certain kind of redundancy: namely those that occur when solving for subsequences starting at a given position. In the next section, we will see, how to improve our bound further.

3.3 Reduction to MCSSE

Algorithm 4.14 (MCSS by Reduction to MCSSE)**.** We can solve the MCSS problem by enumerating all instances of the MCSSE problem and selecting the maximum.

$$
\begin{array}{l}
MCSSReducedForce2\ a = \\
\quad reduce\ max\ -\infty\ \langle\, (MCSSEOpt\ a\ i) : 0 \leq i < |a| \,\rangle
\end{array}
$$

This algorithm has $O(n^2)$ work and $O(\lg n)$ span.

The two algorithms obtained by reduction to MCSSS and reduction to MCSSE both reduce some of the redundant work, but not all, because they don't reduce redundancies when solving for subsequences ending (or starting) at different positions. Next, we will see, how to eliminate these redundant computations.

Lemma 4.1 (MCSSE Extension)**.** Suppose that we are given the maximum contiguous sequence, M_i ending at position i. We can compute the maximum contiguous sequence ending at position $i+1$, M_{i+1}, from this by noticing that

- $M_{i+1} = M_i +\!\!+ \langle\, a[i] \,\rangle$, or
- $M_{i+1} = \langle\, a[i] \,\rangle$,

depending on the sum for each.

Exercise 4.2. Prove the MCSSE Extension lemma .

The algorithm for solving MCSS by reduction to MCSSE solves many instances of MCSSE in parallel. If we give up some parallelism, it turns out that we can improve the work efficiency further based on the MCSSE Extension lemma . The idea is to iterate over the sequence and solve the MCSSE problem for each ending position. To solve the MCSSE problem, we then take the maximum over all positions.

Algorithm 4.15 (MCSS with Iteration)**.** The SPARC code for the algorithm for MCSS obtained by reduction to MCSSE is shown below. We use the function $iteratePrefixes$ to iterate over the input sequence and construct a sequence whose i^{th} position contains the solution to the MCSSE problem at that position.

```
MCSSIterative a =
  let
    f (sum, x) =
      if sum + x ≥ x then
        sum + x
      else
        x
    b = iteratePrefixes f −∞ a
  in
    reduce max −∞ b
  end
```

Cost Analysis. Using array sequences, $iterate Prefixes$ and $reduce$ we are both linear work, because the functions f and max both perform constant work. Because of $iterate Prefixes$, the span is also linear.

Algorithm 4.16 (MCSS: Work Optimal and Low Span)**.** In our $scan$-based algorithm for MCSSE , we used the observation that the maximal contiguous subsequence ending at a given position is identified by subtracting the prefix at the ending position from the minimum sum over all preceeding prefixes. Our new algorithm, uses the same intuition but refines it further by noticing that

- we can compute the sum for all prefixes in one scan, and
- we can compute the minimum prefix sum preceeding all positions in one scan.

After computing these quantities, all that remains is to take the difference and select the maximum.

$$
\begin{array}{l}
MCSSOpt\ a = \\
\quad \texttt{let} \\
\quad\quad (b, v) = scan\ '+'\ 0\ a \\
\quad\quad c = append\ b\ \langle\, v \,\rangle \\
\quad\quad (d, \_) = scan\ min\ \infty\ c \\
\quad\quad e = \langle\, c[i] - d[i] : 0 < i < |a| \,\rangle \\
\quad \texttt{in} \\
\quad\quad reduce\ max\ -\infty\ e \\
\quad \texttt{end}
\end{array}
$$

Example 4.5. Consider the sequence a

$$a = \langle\, 1, -2, 0, 3, -1, 0, 2, -3 \,\rangle\,.$$

Compute

$$
\begin{array}{lcl}
(b, v) & = & scan + 0\ a \\
c & = & append\ b\ \langle\, v \,\rangle\,.
\end{array}
$$

We have $c = \langle\, 0, 1, -1, -1, 2, 1, 1, 3, 0 \,\rangle$.

The sequence c contains the prefix sums ending at each position, including the element at the position; it also contains the empty prefix.

Using the sequence c, we can find the minimum prefix up to all positions as

$$(d, \_) = scan\ min\ \infty\ c$$

to obtain

$$d = \langle \infty, 0, 0, -1, -1 - 1, -1, -1, -1 \rangle .$$

We can now find the maximum subsequence ending at any position i by subtracting the value for i in c from the value for all the prior prefixes calculated in d.

Compute

$$\begin{aligned} e &= \langle c[i] - d[i] : 0 < i < |a| \rangle \\ &= \langle 1, -1, 0, 3, 2, 2, 4, 1 \rangle . \end{aligned}$$

It is not difficult to verify in this small example that the values in e are indeed the maximum contiguous subsequences ending in each position of the original sequence. Finally, we take the maximum of all the values is e to compute the result

$$reduce\ max\ -\infty\ e = 4.$$

Cost of the Algorithm. Using array sequences, and the fact that addition and minimum take constant work, the algorithm performs $\Theta(n)$ work in $\Theta(\lg n)$ span. The algorithm is work optimal because any algorithm must inspect each and every element of the sequence to solve the MCSS problem.

4 Divide And Conquer

4.1 A First Solution

Dividing the Input. To apply divide and conquer, we first need to figure out how to divide the input. There are many possibilities, but dividing the input in two halves is usually a good starting point, because it reduces the input for both subproblems equally, reducing thus the size of the largest component, which is important in bounding the overall span. Correctness is usually independent of the particular strategy of division.

Let us divide the sequence into two halves, recursively solve the problem on both parts, and combine the solutions to solve the original problem.

Example 4.6. Let $a = \langle 1, -2, 0, 3, -1, 0, 2, -3 \rangle$. By using the approach, we divide the sequence into two sequences b and c as follows

$$b = \langle 1, -2, 0, 3 \rangle$$

and

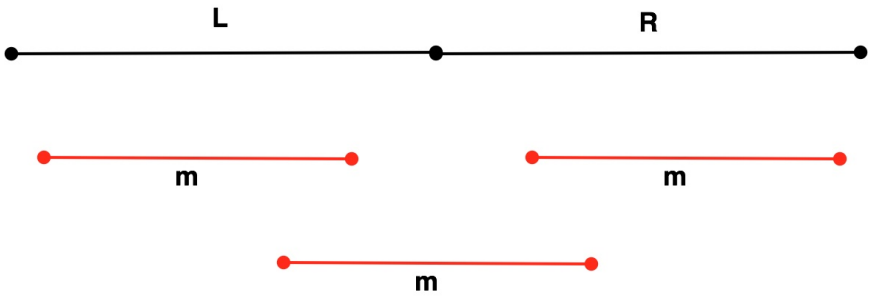
$$c = \langle -1, 0, 2, -3 \rangle$$

We can now solve each part to obtain 3 and 2 as the solutions to the subproblems. Note that there are multiple sequences that yield the maximum sum.

Using Solutions to Subproblems. To construct a solution for the original problem from those of the subproblems, let's consider where the solution subsequence might come from. There are three possibilities.

1. The maximum sum lies completely in the left subproblem.
2. The maximum sum lies completely in the right subproblem.
3. The maximum sum overlaps with both halves, spanning the cut.

The three cases are illustrated below



The first two cases are already solved by the recursive calls, but not the last. Assuming we can find the largest subsequence that spans the cut, we can write our algorithm as shown below.

Algorithm 4.17 (Simple Divide-and-Conquer for MCSS)**.** Using a function called $bestAcross$ to find the largest subsequence that spans the cut, we can write our algorithm as follows.

```
MCSSDC a =
  if |a| = 0 then
    −∞
  else if |a| = 1 then
    a[0]
  else
    let
      (b, c) = splitMid a
      (m_b, m_c) = (MCSSDC b || MCSSDC c)
      m_bc = bestAcross (b, c)
    in
      max{m_b, m_c, m_bc}
    end
```

Algorithm 4.18 (Maximum Subsequence Spanning the Cut)**.** We can reduce the problem of finding the maximum subsequence spanning the cut to two problems that we have seen

already: Maximum-Contiguous-Subsequence Sum with Start, MCSSS, and Maximum-Contiguous-Subsequence Sum at Ending, MCSSE. The maximum sum spanning the cut is the sum of the largest suffix on the left plus the largest prefix on the right. The prefix of the right part is easy as it directly maps to the solution of MCSSS problem at position 0. Similarly, the suffix for the left part is exactly an instance of MCSSE problem. We can thus use the algorithms that we have seen in the previous section for solving this problem, Algorithm **??**, and, Algorithm **??** respectively.

The cost of both of these algorithms is $\Theta(n)$ work and $\Theta(\lg n)$ span and thus the total cost is also the same.

Example 4.7. In the example above, the largest suffix on the left is 3, which is given by the sequence $\langle\, 3 \,\rangle$ or $\langle\, 0, 3 \,\rangle$.

The largest prefix on the right is 1 given by the sequence $\langle\, -1, 0, 2 \,\rangle$. Therefore the largest sum that crosses the middle is $3 + 1 = 4$.

4.1.1 Correctness

To prove a divide-and-conquer algorithm correct, we use the technique of strong induction, which enables to assume that correctness remains correct for all smaller subproblems. We now present such a correctness proof for the algorithm $MCSSDC$.

Theorem 4.2 (Correctness of the algorithm $MCSSDC$)**.** Let a be a sequence. The algorithm $MCSSDC$ returns the maximum contiguous subsequence sum in a gives sequence—and returns $-\infty$ if a is empty.

Proof. The proof will be by (strong) induction on length of the input sequence. Our induction hypothesis is that the theorem above holds for all inputs smaller than the current input.

We have two base cases: one when the sequence is empty and one when it has one element. On the empty sequence, the algorithm returns $-\infty$ and thus the theorem holds. On any singleton sequence $\langle\, x \,\rangle$, the MCSS is x, because

$$\max\left\{\sum_{k=i}^{j} a[k] \;:\; 0 \le i < 1, 0 \le j < 1\right\} = \sum_{k=0}^{0} a[0] = a[0] = x\,.$$

The theorem therefore holds.

For the inductive step, let a be a sequence of length $n \ge 1$, and assume inductively that for any sequence a' of length $n' < n$, the algorithm correctly computes the maximum contiguous subsequence sum. Now consider the sequence a and let b and c denote the left and right subsequences resulted from dividing a into two parts (i.e., $(b, c) = splitMida$). Furthermore, let $a[i \cdots j]$ be any contiguous subsequence of a that has the largest sum, and this value is v. Note that the proof has to account for the possibility that there may be many other subsequences with equal sum. Every contiguous subsequence must start somewhere and end after it. We consider the following 3 possibilities corresponding to how the sequence $a[i \cdots j]$ lies with respect to b and c:

- If the sequence $a[i \cdots j]$ starts in b and ends c. Then its sum equals its part in b (a suffix of b) and its part in c (a prefix of c). If we take the maximum of all suffixes in c and prefixes in b and add them this is equal the maximum of all contiguous sequences bridging the two, because $\max\{x + y : x \in X, y \in Y\} = \max\{x \in X\} + \max\{y \in Y\}$. By assumption this equals the sum of $a[i \cdots j]$ which is v. Furthermore by induction m_b and m_c are sums of other subsequences so they cannot be any larger than v and hence $\max\{m_b, m_c, m_{bc}\} = v$.
- If $a[i \cdots j]$ lies entirely in b, then it follows from our inductive hypothesis that $m_b = v$. Furthermore m_c and m_{bc} correspond to the maximum sum of other subsequences, which cannot be larger than v. So again $\max\{m_b, m_c, m_{bc}\} = v$.
- Similarly, if $a_{i..j}$ lies entirely in c, then it follows from our inductive hypothesis that $m_c = \max\{m_b, m_c, m_{bc}\} = v$.

We conclude that in all cases, we return $\max\{m_b, m_c, m_{bc}\} = v$, as claimed. □

4.1.2 Cost Analysis

By Algorithm **??**, we know that the maximum subsequence crossing the cut in $\Theta(n)$ work and $\Theta(\lg n)$ span. Note also that *splitMid* requires $O(1)$ work and span for array sequences and $O(\lg n)$ work and span for tree sequences, so in either case the work and span are bounded by $O(\lg n)$. We thus have the following recurrences with array-sequence or tree-sequence specifications

$$W(n) = 2W(n/2) + \Theta(n)$$
$$S(n) = S(n/2) + \Theta(\lg n).$$

Using the definition of big-Θ, we know that

$$W(n) \leq 2W(n/2) + k_1 \cdot n + k_2,$$

where k_1 and k_2 are constants. By using the tree method, we can conclude that $W(n) = \Theta(n \lg n)$ and $S(n) = \lg^2 n$.

Solving the Recurrence Using Substitution Method. Let's now redo the recurrences above using the substitution method. Specifically, we'll prove the following theorem using (strong) induction on n.

Theorem 4.3. Let a constant $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot n$ for $n > 1$ and $W(1) \leq k$ for $n \leq 1$, then we can find constants κ_1 and κ_2 such that

$$W(n) \ \leq \ \kappa_1 \cdot n \lg n + \kappa_2.$$

Proof. Let $\kappa_1 = 2k$ and $\kappa_2 = k$. For the base case ($n = 1$), we check that $W(1) = k \leq \kappa_2$. For the inductive step ($n > 1$), we assume that

$$W(n/2) \leq \kappa_1 \cdot \tfrac{n}{2} \lg(\tfrac{n}{2}) + \kappa_2,$$

And we'll show that $W(n) \leq \kappa_1 \cdot n \lg n + \kappa_2$. To show this, we substitute an upper bound for $W(n/2)$ from our assumption into the recurrence, yielding

$$\begin{aligned} W(n) &\leq 2W(n/2) + k \cdot n \\ &\leq 2(\kappa_1 \cdot \tfrac{n}{2} \lg(\tfrac{n}{2}) + \kappa_2) + k \cdot n \\ &= \kappa_1 n(\lg n - 1) + 2\kappa_2 + k \cdot n \\ &= \kappa_1 n \lg n + \kappa_2 + (k \cdot n + \kappa_2 - \kappa_1 \cdot n) \\ &\leq \kappa_1 n \lg n + \kappa_2, \end{aligned}$$

where the final step follows because $k \cdot n + \kappa_2 - \kappa_1 \cdot n \leq 0$ as long as $n > 1$. □

4.2 Divide And Conquer with Strengthening

Our first divide-and-conquer algorithm performs $O(n \lg n)$ work, which is $O(\lg n)$ factor more than the optimal. In this section, we shall reduce the work to $O(n)$ by being more careful about avoiding redundant work.

Intuition. Our divide-and-conquer algorithm has an important redundancy: the maximum prefix and maximum suffix are computed recursively to solve the subproblems for the two halves but are computed again at the combine step of the divide-and-conquer algorithm.

Because these are computed as part of solving the subproblems, we could return them from the recursive calls. To do this, we will strengthen the problem so that it returns the maximum prefix and suffix. This problem, which we shall call ***MCSSPS***, matches the original MCSS problem in its input and returns strictly more information. Solving MCSS using MCSSPS is therefore trivial. We thus focus on the MCSSPS problem.

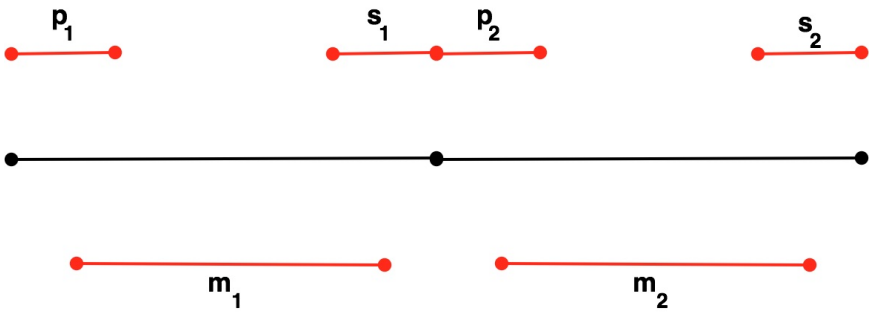
Solving MCSSPS. We can solve this problem by strengthening our divide-and-conquer algorithm from the previous section. We need to return a total of three values:

- the max subsequence sum,
- the max prefix sum, and
- the max suffix sum.

At the base cases, when the sequence is empty or consists of a single element, this is easy to do. For the recursive case, we need to consider how to produce the desired return values from those of the subproblems. Suppose that the two subproblems return (m_1, p_1, s_1) and (m_2, p_2, s_2).

One possibility to compute as result

$$(\max(s_1 + p_2, m_1, m_2), p_1, s_2).$$



Note that we don't have to consider the case when s_1 or p_2 is the maximum, because that case is checked in the computation of m_1 and m_2 by the two subproblems.

This solution fails to account for the case when the suffix and prefix can span the whole sequence.

We can fix this problem by returning the total for each subsequence so that we can compute the maximum prefix and suffix correctly. Thus, we need to return a total of four values:

- the max subsequence sum,
- the max prefix sum,
- the max suffix sum, and
- the overall sum.

Having this information from the subproblems is enough to produce a similar answer tuple for all levels up, in constant work and span per level. Thus what we have discovered is that to solve the strengthened problem efficiently we have to strengthen the problem once again. Thus if the recursive calls return (m_1, p_1, s_1, t_1) and (m_2, p_2, s_2, t_2), then we return

$$(\max(s_1 + p_2, m_1, m_2), \max(p_1, t_1 + p_2), \max(s_1 + t_2, s_2), t_1 + t_2).$$

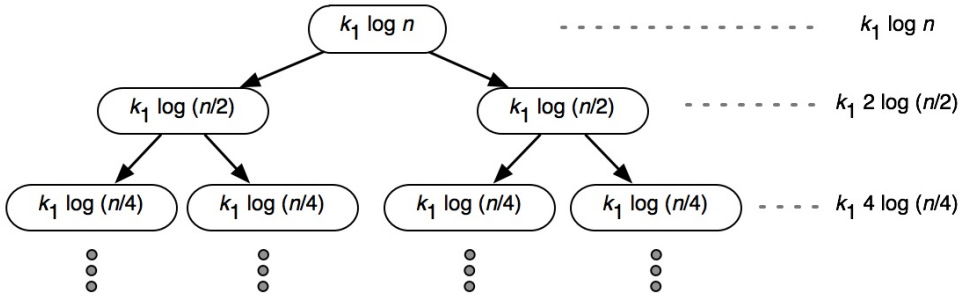
Algorithm 4.19 (Linear Work Divide-and-Conquer MCSS)**.**

```
MCSSDCAux a =
  if |a| = 0 then
    (−∞, −∞, −∞, 0)
  else if |a| = 1then
    (a[0], a[0], a[0], a[0])
  else
    let
      (b, c) = splitMid a
      ((m1, p1, s1, t1), (m2, p2, s2, t2)) = (MCSSDCAux b || MCSSDCAux c)
    in
      (max (s1 + p2, m1, m2),
       max (p1, t1 + p2),
       max (s1 + t2, s2),
       t1 + t2)
    end
MCSSDC a =
  let
    (m, _, _, _) = MCSSDCAux a
  in
    m
  end
```

Cost Analysis. Since *splitMid* requires $O(\lg n)$ work and span in both array and tree sequences, we have

$$W(n) = 2W(n/2) + O(\lg n)$$
$$S(n) = S(n/2) + O(\lg n).$$

The $O(\lg n)$ bound on *splitMid* is not tight for array sequences, where *splitMid* requires $O(1)$ work, but this loose upper bound suffices to achieve the bound on the work that we seek. Note that the span is the same as before, so we'll focus on analyzing the work. Using the tree method, we have



Therefore, the total work is upper-bounded by

$$W(n) \quad \leq \quad \sum_{i=0}^{\lg n} k_1 2^i \lg(n/2^i)$$

It is not so obvious to what this sum evaluates, but we can bound it as follows:

$$\begin{aligned} W(n) &\leq \sum_{i=0}^{\lg n} k_1 2^i \lg(n/2^i) \\ &= \sum_{i=0}^{\lg n} k_1 \left(2^i \lg n - i \cdot 2^i\right) \\ &= k_1 \left(\sum_{i=0}^{\lg n} 2^i\right) \lg n - k_1 \sum_{i=0}^{\lg n} i \cdot 2^i \\ &= k_1(2n-1)\lg n - k_1 \sum_{i=0}^{\lg n} i \cdot 2^i. \end{aligned}$$

We're left with evaluating $s = \sum_{i=0}^{\lg n} i \cdot 2^i$. Observe that if we multiply s by 2, we have

$$2s = \sum_{i=0}^{\lg n} i \cdot 2^{i+1} = \sum_{i=1}^{1+\lg n} (i-1)2^i,$$

so then

$$\begin{aligned} s = 2s - s \; &= \; \sum_{i=1}^{1+\lg n} (i-1)2^i - \sum_{i=0}^{\lg n} i \cdot 2^i \\ &= \left((1+\lg n) - 1\right) 2^{1+\lg n} - \sum_{i=1}^{\lg n} 2^i \\ &= 2n \lg n - (2n-2). \end{aligned}$$

Substituting this back into the expression we derived earlier, we have $W(n) \leq k_1(2n-1)\lg n - 2k_1(n\lg n - n + 1) \in O(n)$ because the $n \lg n$ terms cancel.

We can solve the recurrency by using substitution method also. We'll make a guess that $W(n) \leq \kappa_1 n - \kappa_2 \lg n - k_3$. More precisely, we prove the following theorem.

Theorem 4.4. Let $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot \lg n$ for $n > 1$ and $W(n) \leq k$ for $n \leq 1$, then we can find constants κ_1, κ_2, and κ_3 such that

$$W(n) \; \leq \; \kappa_1 \cdot n - \kappa_2 \cdot \lg n - \kappa_3.$$

Proof. Let $\kappa_1 = 3k$, $\kappa_2 = k$, $\kappa_3 = 2k$. We begin with the base case. Clearly, $W(1) = k \leq \kappa_1 - \kappa_3 = 3k - 2k = k$. For the inductive step, we substitute the inductive hypothesis into

the recurrence and obtain

$$\begin{aligned} W(n) &\le 2W(n/2) + k \cdot \lg n \\ &\le 2(\kappa_1 \tfrac{n}{2} - \kappa_2 \lg(n/2) - \kappa_3) + k \cdot \lg n \\ &= \kappa_1 n - 2\kappa_2(\lg n - 1) - 2\kappa_3 + k \cdot \lg n \\ &= (\kappa_1 n - \kappa_2 \lg n - \kappa_3) + (k \lg n - \kappa_2 \lg n + 2\kappa_2 - \kappa_3) \\ &\le \kappa_1 n - \kappa_2 \lg n - \kappa_3, \end{aligned}$$

where the final step uses the fact that $(k \lg n - \kappa_2 \lg n + 2\kappa_2 - \kappa_3) = (k \lg n - k \lg n + 2k - 2k) = 0 \le 0$ by our choice of κ's. □