CPS 230

# DESIGN AND ANALYSIS OF ALGORITHMS

Fall 2008

Instructor: **Herbert Edelsbrunner**
Teaching Assistant: **Zhiqiang Gu**

# Table of Contents

# 1 Introduction

**Meetings.** We meet twice a week, on Tuesdays and Thursdays, from 1:15 to 2:30pm, in room D106 LSRC.

**Communication.** The course material will be delivered in the two weekly lectures. A written record of the lectures will be available on the web, usually a day after the lecture. The web also contains other information, such as homework assignments, solutions, useful links, etc. The main supporting text is

TARJAN. *Data Structures and Network Algorithms.* SIAM, 1983.

The book focuses on fundamental data structures and graph algorithms, and additional topics covered in the course can be found in the lecture notes or other texts in algorithms such as

KLEINBERG AND TARDOS. *Algorithm Design.* Pearson Education, 2006.

**Examinations.** There will be a final exam (covering the material of the entire semester) and a midterm (at the beginning of October), You may want to freshen up your math skills before going into this course. The weighting of exams and homework used to determine your grades is

| | |
|---|---|
| homework | 35%, |
| midterm | 25%, |
| final | 40%. |

**Homework.** We have seven homeworks scheduled throughout this semester, one per main topic covered in the course. The solutions to each homework are due one and a half weeks after the assignment. More precisely, they are due at the beginning of the third lecture after the assignment. The seventh homework may help you prepare for the final exam and solutions will not be collected.

Rule 1. The solution to any one homework question must fit on a single page (together with the statement of the problem).

Rule 2. The discussion of questions and solutions before the due date is not discouraged, but you must formulate your own solution.

Rule 3. The deadline for turning in solutions is 10 minutes after the beginning of the lecture on the due date.

**Overview.** The main topics to be covered in this course are

I Design Techniques;
II Searching;
III Prioritizing;
IV Graph Algorithms;
V Topological Algorithms;
VI Geometric Algorithms;
VII NP-completeness.

The emphasis will be on algorithm **design** and on algorithm **analysis**. For the analysis, we frequently need basic mathematical tools. Think of analysis as the measurement of the quality of your design. Just like you use your sense of taste to check your cooking, you should get into the habit of using algorithm analysis to justify design decisions when you write an algorithm or a computer program. This is a necessary step to reach the next level in mastering the art of programming. I encourage you to implement new algorithms and to compare the experimental performance of your program with the theoretical prediction gained through analysis.

# I DESIGN TECHNIQUES

## 2 Divide-and-Conquer

We use quicksort as an example for an algorithm that follows the divide-and-conquer paradigm. It has the reputation of being the fasted comparison-based sorting algorithm. Indeed it is very fast on the average but can be slow for some input, unless precautions are taken.

**The algorithm.** Quicksort follows the general paradigm of divide-and-conquer, which means it **divides** the unsorted array into two, it **recurses** on the two pieces, and it finally **combines** the two sorted pieces to obtain the sorted array. An interesting feature of quicksort is that the divide step separates small from large items. As a consequence, combining the sorted pieces happens automatically without doing anything extra.

```
void QUICKSORT(int ℓ, r)
  if ℓ < r then m = SPLIT(ℓ, r);
                QUICKSORT(ℓ, m − 1);
                QUICKSORT(m + 1, r)
  endif.
```

We assume the items are stored in $A[0..n-1]$. The array is sorted by calling QUICKSORT$(0, n-1)$.

**Splitting.** The performance of quicksort depends heavily on the performance of the split operation. The effect of splitting from $\ell$ to $r$ is:

- $x = A[\ell]$ is moved to its correct location at $A[m]$;
- no item in $A[\ell..m-1]$ is larger than $x$;
- no item in $A[m+1..r]$ is smaller than $x$.

Figure 1 illustrates the process with an example. The nine items are split by moving a pointer $i$ from left to right and another pointer $j$ from right to left. The process stops when $i$ and $j$ cross. To get splitting right is a bit delicate, in particular in special cases. Make sure the algorithm is correct for (i) $x$ is smallest item, (ii) $x$ is largest item, (iii) all items are the same.

```
int SPLIT(int ℓ, r)
  x = A[ℓ]; i = ℓ; j = r + 1;
  repeat repeat i++ until x ≤ A[i];
         repeat j-- until x ≥ A[j];
         if i < j then SWAP(i, j) endif
  until i ≥ j;
  SWAP(ℓ, j); return j.
```

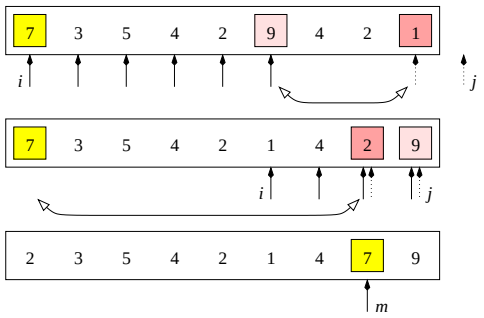


Figure 1: First, $i$ and $j$ stop at items 9 and 1, which are then swapped. Second, $i$ and $j$ cross and the pivot, 7, is swapped with item 2.

Special cases (i) and (iii) are ok but case (ii) requires a stopper at $A[r+1]$. This stopper must be an item at least as large as $x$. If $r < n-1$ this stopper is automatically given. For $r = n-1$, we create such a stopper by setting $A[n] = +\infty$.

**Running time.** The actions taken by quicksort can be expressed using a binary tree: each (internal) node represents a call and displays the length of the subarray; see Figure 2. The worst case occurs when $A$ is already sorted.

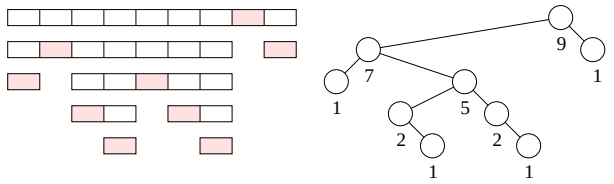


Figure 2: The total amount of time is proportional to the sum of lengths, which are the numbers of nodes in the corresponding subtrees. In the displayed case this sum is 29.

In this case the tree degenerates to a list without branching. The sum of lengths can be described by the following recurrence relation:

$$T(n) \;=\; n + T(n-1) \;=\; \sum_{i=1}^{n} i \;=\; \binom{n+1}{2}.$$

The running time in the worst case is therefore in $O(n^2)$.

In the best case the tree is completely balanced and the sum of lengths is described by the recurrence relation

$$T(n) \;=\; n + 2 \cdot T\left(\frac{n-1}{2}\right).$$

If we assume $n = 2^k - 1$ we can rewrite the relation as

$$\begin{aligned} U(k) &= (2^k - 1) + 2 \cdot U(k-1) \\ &= (2^k - 1) + 2(2^{k-1} - 1) + \ldots + 2^{k-1}(2-1) \\ &= k \cdot 2^k - \sum_{i=0}^{k-1} 2^i \\ &= 2^k \cdot k - (2^k - 1) \\ &= (n+1) \cdot \log_2(n+1) - n. \end{aligned}$$

The running time in the best case is therefore in $\mathrm{O}(n \log n)$.

**Randomization.** One of the drawbacks of quicksort, as described until now, is that it is slow on rather common almost sorted sequences. The reason are pivots that tend to create unbalanced splittings. Such pivots tend to occur in practice more often than one might expect. Human and often also machine generated data is frequently biased towards certain distributions (in this case, permutations), and it has been said that 80% of the time or more, sorting is done on either already sorted or almost sorted files. Such situations can often be helped by transferring the algorithm's dependence on the input data to internally made random choices. In this particular case, we use randomization to make the choice of the pivot independent of the input data. Assume $\textsc{Random}(\ell, r)$ returns an integer $p \in [\ell, r]$ with uniform probability:

$$\mathrm{Prob}[\textsc{Random}(\ell, r) = p] = \frac{1}{r - \ell + 1}$$

for each $\ell \le p \le r$. In other words, each $p \in [\ell, r]$ is equally likely. The following algorithm splits the array with a random pivot:

```
int rSplit(int ℓ, r)
  p = Random(ℓ, r); Swap(ℓ, p);
  return Split(ℓ, r).
```

We get a *randomized* implementation by substituting rSplit for Split. The behavior of this version of quicksort depends on $p$, which is produced by a random number generator.

**Average analysis.** We assume that the items in $A[0..n-1]$ are pairwise different. The pivot splits $A$ into

$$A[0..m-1], \quad A[m], \quad A[m+1..n-1].$$

By assumption on function rSplit, the probability for each $m \in [0, n-1]$ is $\frac{1}{n}$. Therefore the average sum of array lengths split by QuickSort is

$$T(n) = n + \frac{1}{n} \cdot \sum_{m=0}^{n-1} \left(T(m) + T(n-m-1)\right).$$

To simplify, we multiply with $n$ and obtain a second relation by substituting $n-1$ for $n$:

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

$$(n-1) \cdot T(n-1) = (n-1)^2 + 2 \cdot \sum_{i=0}^{n-2} T(i). \qquad (2)$$

Next we subtract (2) from (1), we divide by $n(n+1)$, we use repeated substitution to express $T(n)$ as a sum, and finally split the sum in two:

$$\begin{aligned} \frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2n-1}{n(n+1)} \\ &= \frac{T(n-2)}{n-1} + \frac{2n-3}{(n-1)n} + \frac{2n-1}{n(n+1)} \\ &= \sum_{i=1}^{n} \frac{2i-1}{i(i+1)} \\ &= 2 \cdot \sum_{i=1}^{n} \frac{1}{i+1} - \sum_{i=1}^{n} \frac{1}{i(i+1)}. \end{aligned}$$

**Bounding the sums.** The second sum is solved directly by transformation to a telescoping series:

$$\begin{aligned} \sum_{i=1}^{n} \frac{1}{i(i+1)} &= \sum_{i=1}^{n} \left(\frac{1}{i} - \frac{1}{i+1}\right) \\ &= 1 - \frac{1}{n+1}. \end{aligned}$$

The first sum is bounded from above by the integral of $\frac{1}{x}$ for $x$ ranging from 1 to $n+1$; see Figure 3. The sum of $\frac{1}{i+1}$ is the sum of areas of the shaded rectangles, and because all rectangles lie below the graph of $\frac{1}{x}$ we get a bound for the total rectangle area:

$$\sum_{i=1}^{n} \frac{1}{i+1} < \int_1^{n+1} \frac{\mathrm{d}x}{x} = \ln(n+1).$$

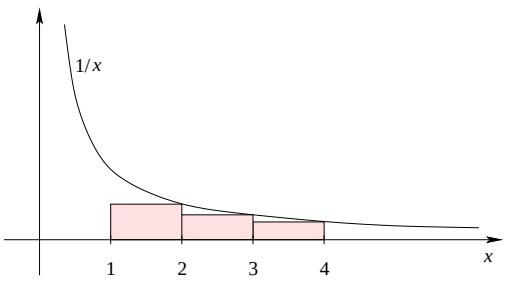


Figure 3: The areas of the rectangles are the terms in the sum, and the total rectangle area is bounded by the integral from 1 through $n+1$.

We plug this bound back into the expression for the average running time:

$$\begin{aligned} T(n) &< (n+1) \cdot \sum_{i=1}^{n} \frac{2}{i+1} \\ &< 2 \cdot (n+1) \cdot \ln(n+1) \\ &= \frac{2}{\log_2 e} \cdot (n+1) \cdot \log_2(n+1). \end{aligned}$$

In words, the running time of quicksort in the average case is only a factor of about $2/\log_2 e = 1.386\ldots$ slower than in the best case. This also implies that the worst case cannot happen very often, for else the average performance would be slower.

**Stack size.** Another drawback of quicksort is the recursion stack, which can reach a size of $\Omega(n)$ entries. This can be improved by always first sorting the smaller side and simultaneously removing the tail-recursion:

```
void QUICKSORT(int ℓ, r)
  i = ℓ; j = r;
  while i < j do
    m = RSPLIT(i, j);
    if m − i < j − m
      then QUICKSORT(i, m − 1); i = m + 1
      else QUICKSORT(m + 1, j); j = m − 1
    endif
  endwhile.
```

In each recursive call to QUICKSORT, the length of the array is at most half the length of the array in the preceding call. This implies that at any moment of time the stack contains no more than $1 + \log_2 n$ entries. Note that without removal of the tail-recursion, the stack can reach $\Omega(n)$ entries even if the smaller side is sorted first.

**Summary.** Quicksort incorporates two design techniques to efficiently sort $n$ numbers: divide-and-conquer for reducing large to small problems and randomization for avoiding the sensitivity to worst-case inputs. The average running time of quicksort is in $\mathrm{O}(n \log n)$ and the extra amount of memory it requires is in $\mathrm{O}(\log n)$. For the deterministic version, the average is over all $n!$ permutations of the input items. For the randomized version the average is the expected running time for *every* input sequence.

# 3 Prune-and-Search

We use two algorithms for selection as examples for the prune-and-search paradigm. The problem is to find the $i$-smallest item in an unsorted collection of $n$ items. We could first sort the list and then return the item in the $i$-th position, but just finding the $i$-th item can be done faster than sorting the entire list. As a warm-up exercise consider selecting the 1-st or smallest item in the unsorted array $A[1..n]$.

```
min = 1;
for j = 2 to n do
  if A[j] < A[min] then min = j endif
endfor.
```

The index of the smallest item is found in $n-1$ comparisons, which is optimal. Indeed, there is an adversary argument, that is, with fewer than $n-1$ comparisons we can change the minimum without changing the outcomes of the comparisons.

**Randomized selection.** We return to finding the $i$-smallest item for a fixed but arbitrary integer $1 \leq i \leq n$, which we call the *rank* of that item. We can use the splitting function of quicksort also for selection. As in quicksort, we choose a random pivot and split the array, but we recurse only for one of the two sides. We invoke the function with the range of indices of the current subarray and the rank of the desired item, $i$. Initially, the range consists of all indices between $\ell = 1$ and $r = n$, limits included.

```
int RSELECT(int ℓ, r, i)
  q = RSPLIT(ℓ, r); m = q − ℓ + 1;
  if i < m then return RSELECT(ℓ, q − 1, i)
    elseif i = m then return q
    else return RSELECT(q + 1, r, i − m)
  endif.
```

For small sets, the algorithm is relatively ineffective and its running time can be improved by switching over to sorting when the size drops below some constant threshold. On the other hand, each recursive step makes some progress so that termination is guaranteed even without special treatment of small sets.

**Expected running time.** For each $1 \leq m \leq n$, the probability that the array is split into subarrays of sizes $m-1$ and $n-m$ is $\frac{1}{n}$. For convenience we assume that $n$ is even. The expected running time increases with increasing number of items, $T(k) \leq T(m)$ if $k \leq m$. Hence,

$$\begin{aligned} T(n) &\leq n + \frac{1}{n}\sum_{m=1}^{n} \max\{T(m-1), T(n-m)\} \\ &\leq n + \frac{2}{n}\sum_{m=\frac{n}{2}+1}^{n} T(m-1). \end{aligned}$$

Assume inductively that $T(m) \leq cm$ for $m < n$ and a sufficiently large positive constant $c$. Such a constant $c$ can certainly be found for $m = 1$, since for that case the running time of the algorithm is only a constant. This establishes the basis of the induction. The case of $n$ items reduces to cases of $m < n$ items for which we can use the induction hypothesis. We thus get

$$\begin{aligned} T(n) &\leq n + \frac{2c}{n}\sum_{m=\frac{n}{2}+1}^{n} m - 1 \\ &= n + c\cdot(n-1) - \frac{c}{2}\cdot\left(\frac{n}{2}+1\right) \\ &= n + \frac{3c}{4}\cdot n - \frac{3c}{2}. \end{aligned}$$

Assuming $c \geq 4$ we thus have $T(n) \leq cn$ as required. Note that we just proved that the expected running time of RSELECT is only a small constant times that of RSPLIT. More precisely, that constant factor is no larger than four.

**Deterministic selection.** The randomized selection algorithm takes time proportional to $n^2$ in the worst case, for example if each split is as unbalanced as possible. It is however possible to select in O($n$) time even in the worst case. The *median* of the set plays a special role in this algorithm. It is defined as the $i$-smallest item where $i = \frac{n+1}{2}$ if $n$ is odd and $i = \frac{n}{2}$ or $\frac{n+2}{2}$ if $n$ is even. The deterministic algorithm takes five steps to select:

`Step 1.` Partition the $n$ items into $\lceil \frac{n}{5} \rceil$ groups of size at most 5 each.

`Step 2.` Find the median in each group.

`Step 3.` Find the median of the medians recursively.

`Step 4.` Split the array using the median of the medians as the pivot.

`Step 5.` Recurse on one side of the pivot.

It is convenient to define $k = \lceil \frac{n}{5} \rceil$ and to partition such that each group consists of items that are multiples of $k$ positions apart. This is what is shown in Figure 4 provided we arrange the items row by row in the array.

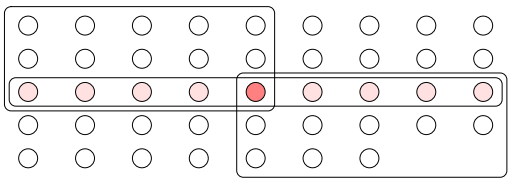

Figure 4: The 43 items are partitioned into seven groups of 5 and two groups of 4, all drawn vertically. The shaded items are the medians and the dark shaded item is the median of medians.

**Implementation with insertion sort.** We use insertion sort on each group to determine the medians. Specifically, we sort the items in positions $\ell, \ell+k, \ell+2k, \ell+3k, \ell+4k$ of array $A$, for each $\ell$.

```
void ISort(int ℓ, k, n)
  j = ℓ + k;
  while j ≤ n do i = j;
    while i > ℓ and A[i] > A[i − k] do
      Swap(i, i − k); i = i − k
    endwhile;
    j = j + k
  endwhile.
```

Although insertion sort takes quadratic time in the worst case, it is very fast for small arrays, as in this application. We can now combine the various pieces and write the selection algorithm in pseudo-code. Starting with the code for the randomized algorithm, we first remove the randomization and second add code for Steps 1, 2, and 3. Recall that $i$ is the rank of the desired item in $A[\ell..r]$. After sorting the groups, we have their medians arranged in the middle fifth of the array, $A[\ell+2k..\ell+3k-1]$, and we compute the median of the medians by recursive application of the function.

```
int Select(int ℓ, r, i)
  k = ⌈(r − ℓ + 1)/5⌉;
  for j = 0 to k − 1 do ISort(ℓ + j, k, r) endfor;
  m′ = Select(ℓ + 2k, ℓ + 3k − 1, ⌊(k + 1)/2⌋);
  Swap(ℓ, m′); q = Split(ℓ, r); m = q − ℓ + 1;
  if i < m then return Select(ℓ, q − 1, i)
    elseif i = m then return q
    else return Select(q + 1, r, i − m)
  endif.
```

Observe that the algorithm makes progress as long as there are at least three items in the set, but we need special treatment of the cases of one or of two items. The role of the median of medians is to prevent an unbalanced split of the array so we can safely use the deterministic version of splitting.

**Worst-case running time.** To simplify the analysis, we assume that $n$ is a multiple of 5 and ignore ceiling and floor functions. We begin by arguing that the number of items less than or equal to the median of medians is at least $\frac{3n}{10}$. These are the first three items in the sets with medians less than or equal to the median of medians. In Figure 4, these items are highlighted by the box to the left and above but containing the median of medians. Symmetrically, the number of items greater than or equal to the median of medians is at least $\frac{3n}{10}$. The first recursion works on a set of $\frac{n}{5}$ medians, and the second recursion works on a set of at most $\frac{7n}{10}$ items. We have

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

We prove $T(n) = \mathrm{O}(n)$ by induction assuming $T(m) \leq c \cdot m$ for $m < n$ and $c$ a large enough constant.

$$\begin{aligned} T(n) &\leq n + \frac{c}{5} \cdot n + \frac{7c}{10} \cdot n \\ &= \left(1 + \frac{9c}{10}\right) \cdot n. \end{aligned}$$

Assuming $c \geq 10$ we have $T(n) \leq cn$, as required. Again the running time is at most some constant times that of splitting the array. The constant is about two and a half times the one for the randomized selection algorithm.

A somewhat subtle issue is the presence of equal items in the input collection. Such occurrences make the function Split unpredictable since they could occur on either side of the pivot. An easy way out of the dilemma is to make sure that the items that are equal to the pivot are treated as if they were smaller than the pivot if they occur in the first half of the array and they are treated as if they were larger than the pivot if they occur in the second half of the array.

**Summary.** The idea of prune-and-search is very similar to divide-and-conquer, which is perhaps the reason why some textbooks make no distinction between the two. The characteristic feature of prune-and-search is that the recursion covers only a constant fraction of the input set. As we have seen in the analysis, this difference implies a better running time.

It is interesting to compare the randomized with the deterministic version of selection:

- the use of randomization leads to a simpler algorithm but it requires a source of randomness;
- upon repeating the algorithm for the same data, the deterministic version goes through the exact same steps while the randomized version does not;
- we analyze the worst-case running time of the deterministic version and the expected running time (for the worst-case input) of the randomized version.

All three differences are fairly universal and apply to other algorithms for which we have the choice between a deterministic and a randomized implementation.

# 4 Dynamic Programming

Sometimes, divide-and-conquer leads to overlapping subproblems and thus to redundant computations. It is not uncommon that the redundancies accumulate and cause an exponential amount of wasted time. We can avoid the waste using a simple idea: **solve each subproblem only once**. To be able to do that, we have to add a certain amount of book-keeping to remember subproblems we have already solved. The technical name for this design paradigm is *dynamic programming*.

**Edit distance.** We illustrate dynamic programming using the edit distance problem, which is motivated by questions in genetics. We assume a finite set of *characters* or *letters*, $\Sigma$, which we refer to as the *alphabet*, and we consider *strings* or *words* formed by concatenating finitely many characters from the alphabet. The *edit distance* between two words is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word to the other. For example, the edit distance between FOOD and MONEY is at most four:

$$\texttt{FOOD} \rightarrow \texttt{MOOD} \rightarrow \texttt{MOND} \rightarrow \texttt{MONED} \rightarrow \texttt{MONEY}$$

A better way to display the editing process is the *gap representation* that places the words one above the other, with a gap in the first word for every insertion and a gap in the second word for every deletion:

```
F O O   D
M O N E Y
```

Columns with two different characters correspond to substitutions. The number of editing steps is therefore the number of columns that do not contain the same character twice.

**Prefix property.** It is not difficult to see that you cannot get from FOOD to MONEY in less than four steps. However, for longer examples it seems considerably more difficult to find the minimum number of steps or to recognize an optimal edit sequence. Consider for example

```
A L G O R   I   T H M
A L   T R U I S T I C
```

Is this optimal or, equivalently, is the edit distance between ALGORITHM and ALTRUISTIC six? Instead of answering this specific question, we develop a dynamic programming algorithm that computes the edit distance between an $m$-character string $A[1..m]$ and an $n$-character string $B[1..n]$. Let $E(i, j)$ be the edit distance between the prefixes of length $i$ and $j$, that is, between $A[1..i]$ and $B[1..j]$. The edit distance between the complete strings is therefore $E(m, n)$. A crucial step towards the development of this algorithm is the following observation about the gap representation of an optimal edit sequence.

PREFIX PROPERTY. If we remove the last column of an optimal edit sequence then the remaining columns represent an optimal edit sequence for the remaining substrings.

We can easily prove this claim by contradiction: if the substrings had a shorter edit sequence, we could just glue the last column back on and get a shorter edit sequence for the original strings.

**Recursive formulation.** We use the Prefix Property to develop a recurrence relation for $E$. The dynamic programming algorithm will be a straightforward implementation of that relation. There are a couple of obvious base cases:

- **Erasing:** we need $i$ deletions to erase an $i$-character string, $E(i, 0) = i$.
- **Creating:** we need $j$ insertions to create a $j$-character string, $E(0, j) = j$.

In general, there are four possibilities for the last column in an optimal edit sequence.

- **Insertion:** the last entry in the top row is empty, $E(i, j) = E(i, j - 1) + 1$.
- **Deletion:** the last entry in the bottom row is empty, $E(i, j) = E(i - 1, j) + 1$.
- **Substitution:** both rows have characters in the last column that are different, $E(i, j) = E(i - 1, j - 1) + 1$.
- **No action:** both rows end in the same character, $E(i, j) = E(i - 1, j - 1)$.

Let $P$ be the logical proposition $A[i] \neq B[j]$ and denote by $|P|$ its indicator variable: $|P| = 1$ if $P$ is true and $|P| = 0$ if $P$ is false. We can now summarize and for $i, j > 0$ get the edit distance as the smallest of the possibilities:

$$E(i, j) \quad = \quad \min \left\{ \begin{array}{l} E(i, j - 1) + 1 \\ E(i - 1, j) + 1 \\ E(i - 1, j - 1) + |P| \end{array} \right\}.$$

**The algorithm.** If we turned this recurrence relation directly into a divide-and-conquer algorithm, we would have the following recurrence for the running time:

$$\begin{aligned} T(m,n) &= T(m,n-1)+T(m-1,n) \\ &\quad + T(m-1,n-1)+1. \end{aligned}$$

The solution to this recurrence is exponential in $m$ and $n$, which is clearly not the way to go. Instead, let us build an $m+1$ times $n+1$ table of possible values of $E(i,j)$. We can start by filling in the base cases, the entries in the 0-th row and column. To fill in any other entry, we need to know the values directly to the left, directly above, and both to the left and above. If we fill the table from top to bottom and from left to right then whenever we reach an entry, the entries it depends on are already available.

```
int EDITDISTANCE(int m, n)
  for i = 0 to m do E[i,0] = i endfor;
  for j = 1 to n do E[0,j] = j endfor;
  for i = 1 to m do
    for j = 1 to n do
      E[i,j] = min{E[i,j-1]+1, E[i-1,j]+1,
                   E[i-1,j-1]+|A[i] ≠ B[j]|}
    endfor
  endfor;
  return E[m,n].
```

Since there are $(m+1)(n+1)$ entries in the table and each takes a constant time to compute, the total running time is in $O(mn)$.

**An example.** The table constructed for the conversion of ALGORITHM to ALTRUISTIC is shown in Figure 5. Boxed numbers indicate places where the two strings have equal characters. The arrows indicate the predecessors that define the entries. Each direction of arrow corresponds to a different edit operation: horizontal for insertion, vertical for deletion, and diagonal for substitution. Dotted diagonal arrows indicate free substitutions of a letter for itself.

**Recovering the edit sequence.** By construction, there is at least one path from the upper left to the lower right corner, but often there will be several. Each such path describes an optimal edit sequence. For the example at hand, we have three optimal edit sequences:

```
A L G O R I   T H M
A L T R U I S T I C
```

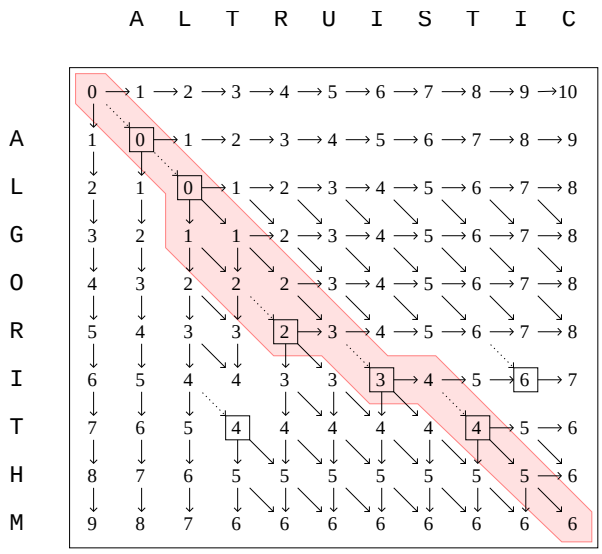


Figure 5: The table of edit distances between all prefixes of ALGORITHM and of ALTRUISTIC. The shaded area highlights the optimal edit sequences, which are paths from the upper left to the lower right corner.

```
A L G O R   I   T H M
A L T   R U I S T I C

A L G O R   I   T H M
A L   T R U I S T I C
```

They are easily recovered by tracing the paths backward, from the end to the beginning. The following algorithm recovers an optimal solution that also minimizes the number of insertions and deletions. We call it with the lengths of the strings as arguments, $\mathrm{R}(m,n)$.

```
void R(int i,j)
  if i > 0 or j > 0 then
    switch incoming arrow:
      case ↘: R(i-1,j-1); print(A[i], B[j])
      case ↓: R(i-1,j); print(A[i], _)
      case →: R(i,j-1); print(_, B[j]).
    endswitch
  endif.
```

**Summary.** The structure of dynamic programming is again similar to divide-and-conquer, except that the subproblems to be solved overlap. As a consequence, we get different recursive paths to the same subproblems. To develop a dynamic programming algorithm that avoids redundant solutions, we generally proceed in two steps:

1. We formulate the problem recursively. In other words, we write down the answer to the whole problem as a combination of the answers to smaller subproblems.
2. We build solutions from bottom up. Starting with the base cases, we work our way up to the final solution and (usually) store intermediate solutions in a table.

For dynamic programming to be effective, we need a structure that leads to at most some polynomial number of different subproblems. Most commonly, we deal with sequences, which have linearly many prefixes and suffixes and quadratically many contiguous substrings.

# 5 Greedy Algorithms

The philosophy of being greedy is shortsightedness. Always go for the seemingly best next thing, always optimize the presence, without any regard for the future, and never change your mind about the past. The greedy paradigm is typically applied to optimization problems. In this section, we first consider a scheduling problem and second the construction of optimal codes.

**A scheduling problem.** Consider a set of activities $1, 2, \ldots, n$. Activity $i$ starts at time $s_i$ and finishes at time $f_i > s_i$. Two activities $i$ and $j$ *overlap* if $[s_i, f_i] \cap [s_j, f_j] \neq \emptyset$. The objective is to select a maximum number of pairwise non-overlapping activities. An example is shown in Figure 6. The largest number of activities can be scheduled by choosing activities with early finish times first. We first sort and reindex such that $i < j$ implies $f_i \leq f_j$.

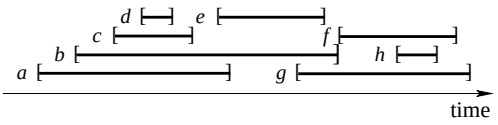


Figure 6: A best schedule is $c, e, f$, but there are also others of the same size.

$S = \{1\}$; $last = 1$;
`for` $i = 2$ `to` $n$ `do`
  `if` $f_{last} < s_i$ `then`
    $S = S \cup \{i\}$; $last = i$
  `endif`
`endfor`.

The running time is O($n \log n$) for sorting plus O($n$) for the greedy collection of activities.

It is often difficult to determine how close to the optimum the solutions found by a greedy algorithm really are. However, for the above scheduling problem the greedy algorithm always finds an optimum. For the proof let $1 = i_1 < i_2 < \ldots < i_k$ be the greedy schedule constructed by the algorithm. Let $j_1 < j_2 < \ldots < j_\ell$ be any other feasible schedule. Since $i_1 = 1$ has the earliest finish time of any activity, we have $f_{i_1} \leq f_{j_1}$. We can therefore add $i_1$ to the feasible schedule and remove at most one activity, namely $j_1$. Among the activities that do not overlap $i_1$, $i_2$ has the earliest finish time, hence $f_{i_2} \leq f_{j_2}$. We can again add $i_2$ to the feasible schedule and remove at most one activity, namely $j_2$ (or possibly $j_1$ if it was not removed before). Eventually, we replace the entire feasible schedule by the greedy schedule without decreasing the number of activities. Since we could have started with a maximum feasible schedule, we conclude that the greedy schedule is also maximum.

**Binary codes.** Next we consider the problem of encoding a text using a string of 0s and 1s. A *binary code* maps each letter in the alphabet of the text to a unique string of 0s and 1s. Suppose for example that the letter 't' is encoded as '001', 'h' is encoded as '101', and 'e' is encoded as '01'. Then the word 'the' would be encoded as the concatenation of codewords: '00110101'. This particular encoding is unambiguous because the code is *prefix-free*: no codeword is prefix of another codeword. There is a one-to-one correspondence between prefix-free binary codes and binary trees where each leaf is a letter and the corresponding codeword is the path from the root to that leaf. Figure 7 illustrates the correspondence for the above 3-letter code. Being prefix-free corresponds to leaves not having children. The tree in Figure 7 is not full because three of its internal nodes have only one child. This is an indication of waste. The code can be improved by replacing each node with one child by its child. This changes the above code to '00' for 't', '1' for 'h', and '01' for 'e'.

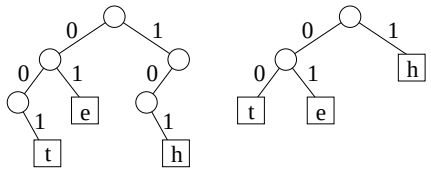


Figure 7: Letters correspond to leaves and codewords correspond to maximal paths. A left edge is read as '0' and a right edge as '1'. The tree to the right is full and improves the code.

**Huffman trees.** Let $w_i$ be the frequency of the letter $c_i$ in the given text. It will be convenient to refer to $w_i$ as the *weight* of $c_i$ or of its external node. To get an efficient code, we choose short codewords for common letters. Suppose $\delta_i$ is the length of the codeword for $c_i$. Then the number of bits for encoding the entire text is

$$P = \sum_i w_i \cdot \delta_i.$$

Since $\delta_i$ is the depth of the leaf $c_i$, $P$ is also known as the *weighted external path length* of the corresponding tree.

The *Huffman tree* for the $c_i$ minimizes the weighted external path length. To construct this tree, we start with $n$ nodes, one for each letter. At each stage of the algorithm, we greedily pick the two nodes with smallest weights and make them the children of a new node with weight equal to the sum of two weights. We repeat until only one node remains. The resulting tree for a collection of nine letters with displayed weights is shown in Figure 8. Ties that

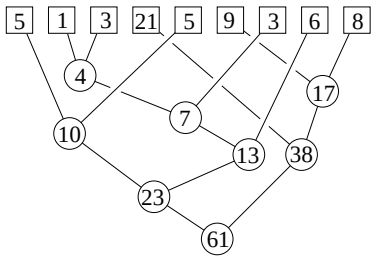


Figure 8: The numbers in the external nodes (squares) are the weights of the corresponding letters, and the ones in the internal nodes (circles) are the weights of these nodes. The Huffman tree is full by construction.

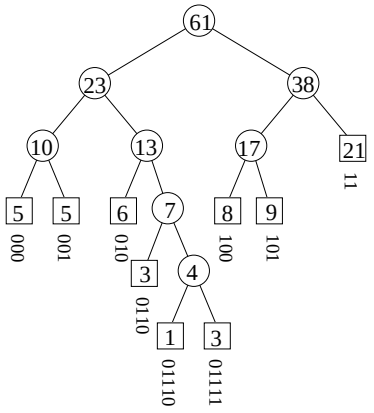


Figure 9: The weighted external path length is $15 + 15 + 18 + 12 + 5 + 15 + 24 + 27 + 42 = 173$.

arise during the algorithm are broken arbitrarily. We redraw the tree and order the children of a node as left and right child arbitrarily, as shown in Figure 9.

The algorithm works with a collection $N$ of nodes which are the roots of the trees constructed so far. Initially, each leaf is a tree by itself. We denote the weight of a node by $w(\mu)$ and use a function EXTRACTMIN that returns the node with the smallest weight and, at the same time, removes this node from the collection.

```
Tree HUFFMAN
  loop μ = EXTRACTMIN(N);
       if N = ∅ then return μ endif;
       ν = EXTRACTMIN(N);
       create node κ with children μ and ν
          and weight w(κ) = w(μ) + w(ν);
       add κ to N
  forever.
```

Straightforward implementations use an array or a linked list and take time O($n$) for each operation involving $N$. There are fewer than $2n$ extractions of the minimum and fewer than $n$ additions, which implies that the total running time is O($n^2$). We will see later that there are better ways to implement $N$ leading to running time O($n \log n$).

**An inequality.** We prepare the proof that the Huffman tree indeed minimizes the weighted external path length. Let $T$ be a full binary tree with weighted external path length $P(T)$. Let $\Lambda(T)$ be the set of leaves and let $\mu$ and $\nu$ be any two leaves with smallest weights. Then we can construct a new tree $T'$ with

(1) set of leaves $\Lambda(T') = (\Lambda(T) - \{\mu, \nu\}) \dot\cup \{\kappa\}$ ,

(2) $w(\kappa) = w(\mu) + w(\nu)$,

(3) $P(T') \leq P(T) - w(\mu) - w(\nu)$, with equality if $\mu$ and $\nu$ are siblings.

We now argue that $T'$ really exists. If $\mu$ and $\nu$ are siblings then we construct $T'$ from $T$ by removing $\mu$ and $\nu$ and declaring their parent, $\kappa$, as the new leaf. Then

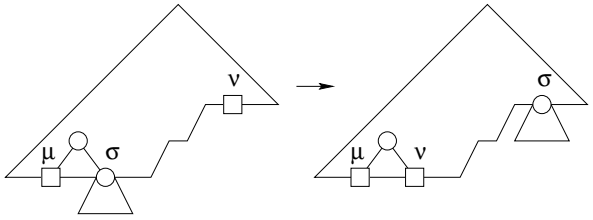


Figure 10: The increase in the depth of $\nu$ is compensated by the decrease in depth of the leaves in the subtree of $\sigma$.

$$\begin{aligned} P(T') &= P(T) - w(\mu)\delta - w(\nu)\delta + w(\kappa)(\delta - 1) \\ &= P(T) - w(\mu) - w(\nu), \end{aligned}$$

where $\delta = \delta(\mu) = \delta(\nu) = \delta(\kappa) + 1$ is the common depth of $\mu$ and $\nu$. Otherwise, assume $\delta(\mu) \geq \delta(\nu)$ and let $\sigma$ be

the sibling of $\mu$, which may or may not be a leaf. Exchange $\nu$ and $\sigma$. Since the length of the path from the root to $\sigma$ is at least as long as the path to $\mu$, the weighted external path length can only decrease; see Figure 10. Then do the same as in the other case.

**Proof of optimality.** The optimality of the Huffman tree can now be proved by induction.

HUFFMAN TREE THEOREM. Let $T$ be the Huffman tree and $X$ another tree with the same set of leaves and weights. Then $P(T) \leq P(X)$.

PROOF. If there are only two leaves then the claim is obvious. Otherwise, let $\mu$ and $\nu$ be the two leaves selected by the algorithm. Construct trees $T'$ and $X'$ with

$$\begin{aligned} P(T') &= P(T) - w(\mu) - w(\nu), \\ P(X') &\leq P(X) - w(\mu) - w(\nu). \end{aligned}$$

$T'$ is the Huffman tree for $n-1$ leaves so we can use the inductive assumption and get $P(T') \leq P(X')$. It follows that

$$\begin{aligned} P(T) &= P(T') + w(\mu) + w(\nu) \\ &\leq P(X') + w(\mu) + w(\nu) \\ &\leq P(X). \end{aligned}$$

□

*Huffman codes* are binary codes that correspond to Huffman trees as described. They are commonly used to compress text and other information. Although Huffman codes are optimal in the sense defined above, there are other codes that are also sensitive to the frequency of sequences of letters and this way outperform Huffman codes for general text.

**Summary.** The greedy algorithm for constructing Huffman trees works bottom-up by stepwise merging, rather than top-down by stepwise partitioning. If we run the greedy algorithm backwards, it becomes very similar to dynamic programming, except that it pursues only one of many possible partitions. Often this implies that it leads to suboptimal solutions. Nevertheless, there are problems that exhibit enough structure that the greedy algorithm succeeds in finding an optimum, and the scheduling and coding problems described above are two such examples.

# First Homework Assignment

Write the solution to each problem on a single page. The deadline for handing in solutions is September 18.

**Problem 1.** (20 points). Consider two sums, $X = x_1 + x_2 + \ldots + x_n$ and $Y = y_1 + y_2 + \ldots + y_m$. Give an algorithm that finds indices $i$ and $j$ such that swapping $x_i$ with $y_j$ makes the two sums equal, that is, $X - x_i + y_j = Y - y_j + x_i$, if they exist. Analyze your algorithm. (You can use sorting as a subroutine. The amount of credit depends on the correctness of the analysis and the running time of your algorithm.)

**Problem 2.** ($20 = 10 + 10$ points). Consider distinct items $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1.0$. The *weighted median* is the item $x_k$ that satisfies

$$\sum_{x_i < x_k} w_i < 0.5 \quad \text{and} \quad \sum_{x_j > x_k} w_j \leq 0.5.$$

(a) Show how to compute the weighted median of $n$ items in worst-case time O($n \log n$) using sorting.

(b) Show how to compute the weighted median in worst-case time O($n$) using a linear-time median algorithm.

**Problem 3.** ($20 = 6 + 14$ points). A game-board has $n$ columns, each consisting of a top number, the cost of visiting the column, and a bottom number, the maximum number of columns you are allowed to jump to the right. The top number can be any positive integer, while the bottom number is either 1, 2, or 3. The objective is to travel from the first column off the board, to the right of the $n$th column. The cost of a game is the sum of the costs of the visited columns.

Assuming the board is represented in a two-dimensional array, $B[2, n]$, the following recursive procedure computes the cost of the cheapest game:

```
int CHEAPEST(int i)
  if i > n then return 0 endif;
  x = B[1,i] + CHEAPEST(i+1);
  y = B[1,i] + CHEAPEST(i+2);
  z = B[1,i] + CHEAPEST(i+3);
  case B[2,i] = 1: return x;
       B[2,i] = 2: return min{x,y};
       B[2,i] = 3: return min{x,y,z}
  endcase.
```

(a) Analyze the asymptotic running time of the procedure.

(b) Describe and analyze a more efficient algorithm for finding the cheapest game.

**Problem 4.** ($20 = 10 + 10$ points). Consider a set of $n$ intervals $[a_i, b_i]$ that cover the unit interval, that is, $[0, 1]$ is contained in the union of the intervals.

(a) Describe an algorithm that computes a minimum subset of the intervals that also covers $[0, 1]$.

(b) Analyze the running time of your algorithm.

(For question (b) you get credit for the correctness of your analysis but also for the running time of your algorithm. In other words, a fast algorithm earns you more points than a slow algorithm.)

**Problem 5.** ($20 = 7 + 7 + 6$ points). Let $A[1..m]$ and $B[1..n]$ be two strings.

(a) Modify the dynamic programming algorithm for computing the edit distance between $A$ and $B$ for the case in which there are only two allowed operations, insertions and deletions of individual letters.

(b) A (not necessarily contiguous) *subsequence* of $A$ is defined by the increasing sequence of its indices, $1 \leq i_1 < i_2 < \ldots < i_k \leq m$. Use dynamic programming to find the longest common subsequence of $A$ and $B$ and analyze its running time.

(c) What is the relationship between the edit distance defined in (a) and the longest common subsequence computed in (b)?

# II SEARCHING

# 6 Binary Search Trees

One of the purposes of sorting is to facilitate fast searching. However, while a sorted sequence stored in a linear array is good for searching, it is expensive to add and delete items. Binary search trees give you the best of both worlds: fast search and fast update.

**Definitions and terminology.** We begin with a recursive definition of the most common type of tree used in algorithms. A *(rooted) binary tree* is either empty or a node (the *root*) with a binary tree as left subtree and binary tree as right subtree. We store items in the nodes of the tree. It is often convenient to say the items *are* the nodes. A binary tree is sorted if each item is between the smaller or equal items in the left subtree and the larger or equal items in the right subtree. For example, the tree illustrated in Figure 11 is sorted assuming the usual ordering of English characters. Terms for relations between family members such as *child*, *parent*, *sibling* are also used for nodes in a tree. Every node has one parent, except the root which has no parent. A *leaf* or *external node* is one without children; all other nodes are *internal*. A node $\nu$ is a *descendent* of $\mu$ if $\nu = \mu$ or $\nu$ is a descendent of a child of $\mu$. Symmetrically, $\mu$ is an *ancestor* of $\nu$ if $\nu$ is a descendent of $\mu$. The *subtree* of $\mu$ consists of all descendents of $\mu$. An *edge* is a parent-child pair.

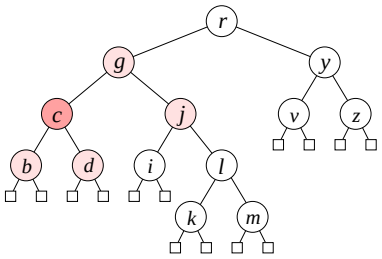


Figure 11: The parent, sibling and two children of the dark node are shaded. The internal nodes are drawn as circles while the leaves are drawn as squares.

The *size* of the tree is the number of nodes. A binary tree is *full* if every internal node has two children. Every full binary tree has one more leaf than internal node. To count its edges, we can either count 2 for each internal node or 1 for every node other than the root. Either way, the total number of edges is one less than the size of the tree. A *path* is a sequence of contiguous edges without repetitions. Usually we only consider paths that descend or paths that ascend. The *length* of a path is the number of edges. For every node $\mu$, there is a unique path from the root to $\mu$. The length of that path is the *depth* of $\mu$. The *height* of the tree is the maximum depth of any node. The *path length* is the sum of depths over all nodes, and the *external path length* is the same sum restricted to the leaves in the tree.

**Searching.** A *binary search tree* is a sorted binary tree. We assume each node is a record storing an item and pointers to two children:

```
struct Node{item info; Node *ℓ, *r};
typedef Node *Tree.
```

Sometimes it is convenient to also store a pointer to the parent, but for now we will do without. We can search in a binary search tree by tracing a path starting at the root.

```
Node *SEARCH(Tree ϱ, item x)
  case ϱ = NULL: return NULL;
       x < ϱ → info: return SEARCH(ϱ → ℓ, x);
       x = ϱ → info: return ϱ;
       x > ϱ → info: return SEARCH(ϱ → r, x)
  endcase.
```

The running time depends on the length of the path, which is at most the height of the tree. Let $n$ be the size. In the worst case the tree is a linked list and searching takes time $O(n)$. In the best case the tree is perfectly balanced and searching takes only time $O(\log n)$.

**Insert.** To add a new item is similarly straightforward: follow a path from the root to a leaf and replace that leaf by a new node storing the item. Figure 12 shows the tree obtained after adding $w$ to the tree in Figure 11. The running time depends again on the length of the path. If the insertions come in a random order then the tree is usually

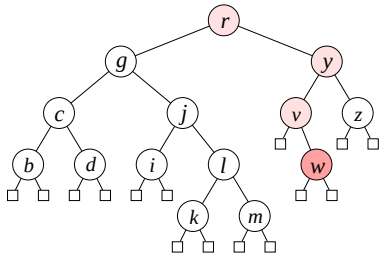


Figure 12: The shaded nodes indicate the path from the root we traverse when we insert $w$ into the sorted tree.

close to being perfectly balanced. Indeed, the tree is the same as the one that arises in the analysis of quicksort. The expected number of comparisons for a (successful) search is one $n$-th of the expected running time of quicksort, which is roughly $2 \ln n$.

**Delete.** The main idea for deleting an item is the same as for inserting: follow the path from the root to the node $\nu$ that stores the item.

`Case 1.` $\nu$ has no internal node as a child. Remove $\nu$.

`Case 2.` $\nu$ has one internal child. Make that child the child of the parent of $\nu$.

`Case 3.` $\nu$ has two internal children. Find the rightmost internal node in the left subtree, remove it, and substitute it for $\nu$, as shown in Figure 13.

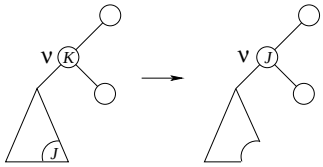


Figure 13: Store $J$ in $\nu$ and delete the node that used to store $J$.

The analysis of the expected search time in a binary search tree constructed by a random sequence of insertions and deletions is considerably more challenging than if no deletions are present. Even the definition of a random sequence is ambiguous in this case.

**Optimal binary search trees.** Instead of hoping the incremental construction yields a shallow tree, we can construct the tree that minimizes the search time. We consider the common problem in which items have different probabilities to be the target of a search. For example, some words in the English dictionary are more commonly searched than others and are therefore assigned a higher probability. Let $a_1 < a_2 < \ldots < a_n$ be the items and $p_i$ the corresponding probabilities. To simplify the discussion, we only consider successful searches and thus assume $\sum_{i=1}^{n} p_i = 1$. The expected number of comparisons for a successful search in a binary search tree $T$ storing the $n$ items is

$$\begin{aligned} 1 + C(T) &= \sum_{i=1}^{n} p_i \cdot (\delta_i + 1) \\ &= 1 + \sum_{i=1}^{n} p_i \cdot \delta_i, \end{aligned}$$

where $\delta_i$ is the depth of the node that stores $a_i$. $C(T)$ is the *weighted path length* or the *cost* of $T$. We study the problem of constructing a tree that minimizes the cost. To develop an example, let $n = 3$ and $p_1 = \frac{1}{2}$, $p_2 = \frac{1}{3}$, $p_3 = \frac{1}{6}$. Figure 14 shows the five binary trees with three nodes and states their costs. It can be shown that the

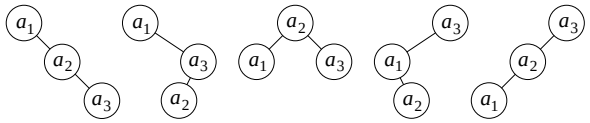


Figure 14: There are five different binary trees of three nodes. From left to right their costs are $\frac{2}{3}, \frac{5}{6}, \frac{2}{3}, \frac{7}{6}, \frac{4}{3}$. The first tree and the third tree are both optimal.

number of different binary trees with $n$ nodes is $\frac{1}{n+1}\binom{2n}{n}$, which is exponential in $n$. This is far too large to try all possibilities, so we need to look for a more efficient way to construct an optimum tree.

**Dynamic programming.** We write $T_i^j$ for the optimum weighted binary search tree of $a_i, a_{i+1}, \ldots, a_j$, $C_i^j$ for its cost, and $p_i^j = \sum_{k=i}^{j} p_k$ for the total probability of the items in $T_i^j$. Suppose we know that the optimum tree stores item $a_k$ in its root. Then the left subtree is $T_i^{k-1}$ and the right subtree is $T_{k+1}^j$. The cost of the optimum tree is therefore $C_i^j = C_i^{k-1} + C_{k+1}^j + p_i^j - p_k$. Since we do not know which item is in the root, we try all possibilities and find the minimum:

$$C_i^j = \min_{i \le k \le j} \{C_i^{k-1} + C_{k+1}^j + p_i^j - p_k\}.$$

This formula can be translated directly into a dynamic programming algorithm. We use three two-dimensional arrays, one for the sums of probabilities, $p_i^j$, one for the costs of optimum trees, $C_i^j$, and one for the indices of the items stored in their roots, $R_i^j$. We assume that the first array has already been computed. We initialize the other two arrays along the main diagonal and add one dummy diagonal for the cost.