Technische Universiteit
Eindhoven
University of Technology

Course Notes – Fall 2013

Advanced Algorithms (2IL45)

Mark de Berg

Contents

1 Introduction to randomized algorithms **4**
1.1 Some probability-theory trivia . . . 4
1.2 Randomized approximate median . . . 5
1.3 The hiring problem . . . 7

2 Selection and sorting **10**
2.1 Selection . . . 10
2.2 Randomized QuickSort . . . 13

3 Randomized search structures **16**
3.1 Treaps . . . 16
3.2 Skip lists . . . 20

4 Randomized permutation routing on the hypercube **23**

5 The probabilistic method **27**
5.1 MaxSat . . . 27
5.2 Independent set . . . 28

6 Introduction to Approximation Algorithms **30**
6.1 Load balancing . . . 31

7 The Traveling Salesman Problem **34**
7.1 A simple 2-approximation algorithm . . . 35
7.2 Christofides's (3/2)-approximation algorithm . . . 37

8 Approximation via LP Rounding **40**
8.1 Unweighted vertex cover . . . 40
8.2 Weighted Vertex Cover . . . 41
8.3 Set Cover . . . 44

9 Polynomial-time approximation schemes **48**
9.1 Knapsack . . . 48

Chapter 1

Introduction to randomized algorithms

A *randomized algorithm* is an algorithm whose working not only depends on the input but also on certain random choices made by the algorithm.

Assumption: We have a random number generator $\mathit{Random}(a,b)$ that generates for two integers a, b with $a < b$ an integer r with $a \leqslant r \leqslant b$ uniformly at random. In other words, $\Pr[r = i] = 1/(b-a+1)$ for all $a \leqslant i \leqslant b$. We assume that $\mathit{Random}(a,b)$ runs in $O(1)$ time (even though it would perhaps be more fair if we could only get a single random bit in $O(1)$ time, and in fact we only have *pseudo-random number generators*).

1.1 Some probability-theory trivia

Running example: flip a coin twice and write down the outcomes (heads or tails).

- Sample space = elementary events = possible outcomes of experiment = $\{HH, HT, TH, TT\}$.
- Events = subsets of sample space. For example $\{HH, HT, TH\}$ (at least one heads).
- Each event has a certain probability. Uniform probability distribution: all elementary events have equal probability: $\Pr[HH] = \Pr[HT] = \Pr[TH] = \Pr[TT] = 1/4$.
- Random variable: assign a number to each elementary event. For example random variable X defined as the number of heads: $X(HH) = 2$, $X(HT) = X(TH) = 1$, $X(TT) = 0$.
- Expected value of (discrete) random variable.

  $$\mathrm{E}[X] = \sum_x x \cdot \Pr[X = x],$$

  with the sum taken over all possible values of X. For the example above:

  $$\mathrm{E}[X] = (1/4) \cdot 2 + (1/4) \cdot 1 + (1/4) \cdot 1 + (1/4) \cdot 0 = 1.$$

- Define some other random variable Y, e.g. $Y(HH) = Y(TT) = 1$ and $Y(TH) = Y(HT) = 0$. We have $\mathrm{E}[Y] = 1/2$. Now consider the expectation of $X + Y$:

  $$\begin{aligned}\mathrm{E}[X+Y] &= \Pr[HH] \cdot (X(HH) + Y(HH)) + \cdots + \Pr[TT] \cdot (X(TT) + Y(TT)) \\ &= (1/4) \cdot (2+1) + (1/4) \cdot (1+0) + (1/4) \cdot (1+0) + (1/4) \cdot (0+1) = 3/2.\end{aligned}$$

  Thus, $\mathrm{E}[X+Y] = \mathrm{E}[X] + \mathrm{E}[Y]$. This is always the case, because of *linearity of expectation:*

  For any two random variables X, Y we have $\mathrm{E}[X+Y] = \mathrm{E}[X] + \mathrm{E}[Y]$.

- Do we also have $\mathrm{E}[X \cdot Y] = \mathrm{E}[X] \cdot \mathrm{E}[Y]$? In general, no. (Check example above: $\mathrm{E}[X^2] = 3/2 \neq 1 = \mathrm{E}[X]^2$.) It is true if the variables are *independent*.

- Let X be a random variable. Then the *Markov Inequality* gives a bound on the probability that the actual value of X is t times larger than its expected value:

  Lemma 1.1 (Markov inequality) *Let X be a non-negative random variable, and $\mu = \mathrm{E}[X]$ be its expectation. Then for any $t > 0$ we have $\Pr[\ X > t \cdot \mu\ ] \leqslant 1/t$.*

- *Bernoulli trial*: experiment with two possible outcomes: success or fail. If the probability of success is p, then the expected number of trials before a successful experiment is $1/p$.

  If we consider experiments with two possible outcomes, 0 or 1, but the success probability is different for each experiment, then these are called *Poisson trials.* The following result is often useful to obtain high-probability bounds for randomized algorithms:

  Lemma 1.2 (Tail estimates for Poisson trials) *Suppose we do n Poisson trials. Let X_i denote the outcome of the i-th trial and let $p_i = \Pr[X_i = 1]$, where $0 < p_i < 1$. Let $X = \sum_{i=1}^{n} X_i$ and let $\mu = \mathrm{E}[X] = \sum_{i=1}^{n} p_i$ be the expected number of successful experiments. Then*

  $$\Pr[X > (1+\delta)\mu] \leqslant \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}} \right)^{\mu}.$$

  Thus the probability of deviating from the expected value decreases exponentially. For example, for $\delta = 2$ we have $e^{\delta}/(1+\delta)^{1+\delta} < 1/2$, so we get

  $$\Pr[X > 3\mu] \leqslant \left(\frac{1}{2} \right)^{\mu}.$$

1.2 Randomized approximate median

Let S be a set of n numbers. Assume for simplicity that all numbers are distinct. The *rank* of a number $x \in S$ is 1 plus the number of elements in S that are smaller than x:

$$\mathit{rank}(x) = 1 + |\{y \in S : y < x\}|.$$

Thus the smallest element has rank 1 and the largest element has rank n (under the assumption that all elements are distinct). A *median* of S is a number of rank $\lfloor (n+1)/2 \rfloor$ or

$\lceil (n+1)/2 \rceil$. Later we will look at the problem of finding a median in S—in fact, we will look at the more general problem of finding an element of a given rank. In many applications, however, we do not need the exact median; a number close to the median is good enough.

A δ-*approximate median* is an element of rank k with $\lfloor (\frac{1}{2}-\delta)(n+1) \rfloor \leqslant k \leqslant \lceil (\frac{1}{2}+\delta)(n+1) \rceil$, for some given constant $0 \leqslant \delta \leqslant 1/2$. The following algorithm tries to find a δ-approximate median in a set of number given in an array $A[1..n]$.

Algorithm *ApproxMedian1* (δ, A)
```
1.  ▷ $A[1..n]$ is array of $n$ distinct numbers.
2.  $r \leftarrow Random(1, n)$
3.  $x^* \leftarrow A[r]$; $k \leftarrow 1$
4.  for $i \leftarrow 1$ to $n$
5.      do if $A[i] < x^*$ then $k \leftarrow k+1$
6.  if $\lfloor (\frac{1}{2}-\delta)(n+1) \rfloor \leqslant k \leqslant \lceil (\frac{1}{2}+\delta)(n+1) \rceil$
7.     then return $x^*$
8.     else return “error”
```

ApproxMedian1 clearly runs in $O(n)$ time. Unfortunately, it does not always correctly solve the problem: when it reports an element x^* this element is indeed a δ-approximate median, but it may also report “error”. When does the algorithm succeed? This happens when the random element that is picked is one of the

$$\lceil (\frac{1}{2}+\delta)(n+1) \rceil - \lfloor (\frac{1}{2}-\delta)(n+1) \rfloor + 1$$

possible δ-approximate medians. Since the index r is chosen uniformly at random in line 2, this happens with probability

$$\frac{\lceil (\frac{1}{2}+\delta)(n+1) \rceil - \lfloor (\frac{1}{2}-\delta)(n+1) \rfloor + 1}{n} \approx 2\delta.$$

For example, for $\delta = 1/4$—thus we are looking for an element of rank between $n/4$ and $3n/4$—the success probability is 1/2. If we are looking for an element that is closer to the median, say $\delta = 1/10$, then things get worse. However, there's an easy way to improve the success rate of the algorithm: we choose a threshold value c, and repeat *ApproxMedian1* until we have found a δ-approximate median or until we have tried c times.

Algorithm *ApproxMedian2* (δ, A)
```
1.  $j \leftarrow 1$
2.  repeat $result \leftarrow ApproxMedian1(A, \delta)$; $j \leftarrow j+1$
3.  until ($result \neq$ “error”) or ($j = c+1$)
4.  return $result$
```

The worst-case running time of this algorithm is $O(cn)$. What is the probability that it fails to report a δ-approximate median? For this to happen, all c trials must fail, which happens with probability (roughly) $(1-2\delta)^c$. So if we want to find a (1/4)-approximate median, then by setting $c = 10$ we obtain a linear-time algorithm whose success rate is roughly 99.9%. Pretty good. Even if we are more demanding and want to have a (1/10)-approximate median, then with $c = 10$ we still obtain a success rate of more than 89.2%.

Of course if we insist on finding a δ-approximate median, then we can just keep on trying until finally *ApproxMedian1* is successful:

```
Algorithm ApproxMedian3($\delta, A$)
1.  repeat result ←ApproxMedian1($\delta, A$)
2.  until result ≠ "error"
3.  return result
```

This algorithm always reports a δ-approximate median—or perhaps we should say: it never produces an incorrect answer or "error"—but its running time may vary. If we are lucky then the first run of *ApproxMedian1* already produces a δ-approximate median. If we are unlucky, however, it may take many many trials before we finally have success. Thus we cannot bound the *worst-case* running time as a function of n. We can say something about the *expected* running time, however. Indeed, as we have seen above, a single run of *ApproxMedian1*(δ, A) is successful with probability (rouhly) 2δ. Hence, the expected number of trials until we have success is $1/(2\delta)$. We can now bound the expected running time of *ApproxMedian3* as follows.

$$\begin{aligned}
\text{E[running time of } \textit{ApproxMedian3} \text{]} &= \text{E[(number of calls to } \textit{ApproxMedian1}) \cdot O(n) \text{]} \\
&= O(n) \cdot \text{E[number of calls to } \textit{ApproxMedian1} \text{]} \\
&= O(n) \cdot (1/2\delta) \\
&= O(n/\delta)
\end{aligned}$$

Thus the algorithm runs in $O(n/\delta)$ expected time. The expectation in the running time has nothing to do with the specific input: the expected running time is $O(n/\delta)$ for *any input*. In other words, we are not assuming anything about the input distribution (in particular, we do not assume that the elements are stored in A in random order): the randomization is under full control of the algorithm. In effect, the algorithm *uses* randomization to ensure that the expected running time is $O(n/\delta)$, no matter in which order the numbers are stored in A. Thus it is actually more precise to speak of the *worst-case expected running time*: for different inputs the expected running time may be different—this is not the case in the *ApproxMedian3*, by the way—and we are interested in the maximum expected running time. As is commonly done, when we will talk about "expected running time" in the sequel we actually mean "worst-case expected running time".

Monte Carlo algorithms and Las Vegas algorithms. Note that randomization shows up in different ways in the algorithms we have seen. In *ApproxMedian1* the random choices made by the algorithm influenced the correctness of the algorithm, but the running time was independent of the random choices. Such an algorithm is called a *Monte Carlo algorithm*. In *ApproxMedian3* on the other hand, the random choices made by the algorithm influenced the running time, but the algorithm always produces a correct solution. Such an algorithm is called a *Las Vegas* algorithm. *ApproxMedian2* is mixture: the random choices influence both the running time and the correctness. Sometimes this is also called a Monte Carlo algorithm.

1.3 The hiring problem

Suppose you are doing a very important project, for which you need the best assistant you can get. You contact an employment agency that promises to send you some candidates, one per day, for interviewing. Since you really want to have the best assistant available, you decide on the following strategy: whenever you interview a candidate and the candidate turns out to be better than your current assistent, you fire your current assistant and hire the new candidate. If the total number of candidates is n, this leads to the following algorithm.

```
Algorithm Hire-Assistant
1.  CurrentAssistant ← nil
2.  for $i \leftarrow 1$ to $n$
3.      do Interview candidate $i$
4.          if candidate $i$ is better than CurrentAssistant
5.              then CurrentAssistant ← candidate $i$
```

Suppose you have to pay a fee of f euro whenever you hire a new candidate (that is, when *CurrentAssistant* changes). In the worst case, every candidate is better than all previous ones, and you end up paying $f \cdot n$ euro to the employment agency. Of course you would like to pay less. One way is to proceed as follows. First, you ask to employment agency to send you the complete list of n candidates. Now you have to decide on an order to interview the candidates. The list itself does not give you any information on the quality of the candidates, only their names, and in fact you do not quite trust the employment agency: maybe they have put the names in such an order that interviewing them in the given order (or in the reverse order) would maximize their profit. Therefore you proceed as follows.

```
Algorithm Hire-Assistant-Randomized
1.  Compute a random permutation of the candidates.
2.  CurrentAssistant ← nil
3.  for $i \leftarrow 1$ to $n$
4.      do Interview candidate $i$ in the random permutation
5.          if candidate $i$ is better than CurrentAssistant
6.              then CurrentAssistant ← candidate $i$
```

What is the expected cost of this algorithm, that is, the expected total fee you have to pay to the employment agency?

$$\begin{aligned}\mathrm{E}[\text{ cost }] &= \mathrm{E}[\ \textstyle\sum_{i=1}^{n} (\text{cost to be paid for } i\text{-th candidate})\ ] \\ &= \textstyle\sum_{i=1}^{n} \mathrm{E}[\ (\text{cost to be paid for } i\text{-th candidate})\ ] \text{ (by linearity of expectation)}\end{aligned}$$

You have to pay f euro when i-th candidate is better than candidates $1, \ldots, i-1$, otherwise you pay nothing. Consider the event "i-th candidate is better than candidates $1, \ldots, i-1$", and introduce *indicator random variable* X_i for it:

$$X_i = \begin{cases} 1 & \text{if } i\text{-th candidate is better than candidates } 1, \ldots, i-1 \\ 0 & \text{otherwise} \end{cases}$$

(An indicator random variable for an event is a variable that is 1 if the event takes place and 0 otherwise.) Now the fee you have to pay for the i-th candidate is $X_i \cdot f$ euro, so we can write

$$\mathrm{E}[\ (\text{fee to be paid for } i\text{-th candidate})\ ] = \mathrm{E}[X_i \cdot f] = f \cdot \mathrm{E}[X_i]$$

Let C_i denote the first i candidates in the random order. C_i itself is also in random order, and so the i-th candidate is the best candidate in C_i with probability $1/i$. Hence,

$$\mathrm{E}[X_i] = \Pr[X_i = 1] = 1/i,$$

and we get

$$\begin{aligned} \mathrm{E}[\text{ total cost }] &= \textstyle\sum_{i=1}^{n} \mathrm{E}[\text{ (fee to be paid for } i\text{-th candidate) }] \\ &= \textstyle\sum_{i=1}^{n} f \cdot \mathrm{E}[X_i] \\ &= f \cdot \textstyle\sum_{i=1}^{n} (1/i) \\ &\approx f \ln n \end{aligned}$$

The algorithm above needs to compute a random permutation on the candidates. Here is an algorithm for computing a random permutation on a set of elements stored in an array A:

```
Algorithm RandomPermutation(A)
1.  ▷ Compute random permutation of array $A[1..n]$
2.  for $i \leftarrow 1$ to $n-1$
3.      do $r \leftarrow Random(i, n)$
4.          Exchange $A[i]$ and $A[r]$
```

Note: The algorithm does not work if we replace $Random(i, n)$ in line 3 by $Random(1, n)$.

Chapter 2

Selection and sorting

2.1 Selection

Given a set S of n distinct numbers and an integer i with $1 \leqslant i \leqslant n$, we want to find the element of rank i in S. The following algorithm solves the problem:

```
Algorithm Select(S, i)
1.  ▷ selects element of rank $i$ from $S$, assuming $1 \leqslant i \leqslant |S|$
2.  if $|S| = 1$
3.     then return the only element in $S$
4.     else  Pick a pivot element $x_{\text{piv}} \in S$.
5.           $S_< \leftarrow \{x \in S : x < x_{\text{piv}}\}$; $S_> \leftarrow \{x \in S : x > x_{\text{piv}}\}$
6.           $k \leftarrow |S_<|$
7.           if $k = i - 1$
8.              then return $x_{\text{piv}}$
9.              else  if $k > i - 1$
10.                      then return Select($S_<, i$)
11.                      else  return Select($S_>, i - k - 1$)
```

In the worst case, the pivot element splits S in a very unbalanced manner and we have to recurse on the larger part. This will happen for example when we are searching for the element of rank n—the largest element—and the pivot is always the smallest element. This leads to the following recurrence on the worst-case running time:

$$T_{\text{worst}}(n) = O(n) + T_{\text{worst}}(n-1),$$

which solves to $T_{\text{worst}}(n) = O(n^2)$. To get a better running time we would like to get a balanced split, so that we never have to recurse on a set with too many elements. For instance, the median would be a very good choice as pivot, because that would give a recurrence of the form

$$T(n) = O(n) + T(n/2),$$

which solves to $O(n)$. Unfortunately finding the median is not easy—in fact, finding an element of a given rank is exactly the problem we were trying to solve in the first place! There is an elegant algorithm that manages to solve the selection problem in $O(n)$ time in the worst case. We will discuss this later. But if we are happy with a randomized algorithm wit $O(n)$ expected running time, then things are pretty easy: all we have to do is to pick the pivot element uniformly at random. Thus we change line 4 of *Select* to

```
4.     else Pick a pivot element $x_{\mathrm{piv}} \in S$ uniformly at random.
```

Next we analyze the expected running time of this algorithm. The idea is, of course, that a random element is expected to be close enough to the median so that a good expected running time results. Now we have to prove that this is indeed the case.

One way to do the analysis is as follows. Note that when the pivot has rank j then we either recurse on a subset of size $j-1$, or we recurse on a subset of size $n-j$, or (if $j=i$) we are done. Thus in the worst case we would recurse on a subset of size $\max(j-1, n-j)$. Now each $1 \leqslant j \leqslant n$ has probability $1/n$ of being the rank of the pivot. Hence, the expected running time $T_{\mathrm{exp}}(n)$ is bounded by

$$\begin{aligned} T_{\mathrm{exp}}(n) &\leqslant O(n) + \textstyle\sum_{j=1}^{n} \Pr[\text{ element of rank } j \text{ is pivot }] \cdot T_{\mathrm{exp}}(\max(j-1, n-j)) \\ &= O(n) + \textstyle\frac{1}{n}\sum_{j=1}^{n} T_{\mathrm{exp}}(\max(j-1, n-j)) \end{aligned}$$

and then show that this recurrence solves to $T_{\mathrm{exp}}(n) = O(n)$. A somewhat easier method is to use the observation that with probability $1/2$ we recurse on at most $3n/4$ elements. Since $T(n)$ is increasing, we get

$$T_{\mathrm{exp}}(n) \leqslant O(n) + \frac{1}{2}T_{\mathrm{exp}}(3n/4) + \frac{1}{2}T_{\mathrm{exp}}(n-1). \tag{2.1}$$

This recurrence is pretty easy to solve by induction, as shown next.

Lemma 2.1 *$T_{\mathrm{exp}}(n) = O(n)$.*

Proof. From Equation (2.1) we conclude that there is a constant c such that

$$T_{\mathrm{exp}}(n) \leqslant cn + \frac{1}{2}T_{\mathrm{exp}}(3n/4) + \frac{1}{2}T_{\mathrm{exp}}(n-1).$$

for all $n > 1$. We claim that there is a constant d such that $T_{\mathrm{exp}}(n) \leqslant dn$ for all $n \geqslant 1$. Since obviously $T_{\mathrm{exp}}(1) = O(1)$, there is a constant c' such that $T_{\mathrm{exp}}(1) = c'$. Take $d = \max(8c, c')$. Then our claim is true for $n = 1$. For $n > 1$ we get

$$\begin{aligned} T_{\mathrm{exp}}(n) &\leqslant cn + \tfrac{1}{2}T_{\mathrm{exp}}(3n/4) + \tfrac{1}{2}T_{\mathrm{exp}}(n-1) \\ &\leqslant cn + \tfrac{1}{2}\cdot d(3n/4) + \tfrac{1}{2}\cdot d(n-1) && \text{(by induction)} \\ &\leqslant cn + \tfrac{7}{8}dn \\ &\leqslant dn && \text{(since } d \geqslant 8c\text{)} \end{aligned}$$

We conclude that $T_{\mathrm{exp}}(n) = O(n)$. □

An alternative approach. Instead of picking the pivot uniformly at random from S, we could also insist on picking a good pivot. An easy way to do this is to use algorithm *ApproxMedian3* (see the Course Notes for Lecture1) to find a (1/4)-approximate median. Now the expected running time is bounded by

$$\text{E}[\ O(n) + (\text{time for } \textit{ApproxMedian3} \text{ with } \delta = 1/4) + (\text{time for recursive call})\ ].$$

The recursive call is now on at most $3n/4$ elements and the expected running time of *ApproxMedian3* is $O(n/\delta) = O(n)$. Hence, we get

$$\begin{aligned} T_{\text{exp}}(n) &= \text{E}[\ O(n) + (\text{time for } \textit{ApproxMedian3} \text{ with } \delta = 1/4) + (\text{time for recursive call})\ ] \\ &\leqslant O(n) + T_{\text{exp}}(3n/4) \end{aligned}$$

which again solves to $O(n)$.

A deterministic solution. The following is a deterministic algorithm that solves the selection problem in $O(n)$ time in the worst case. It is the same as the algorithm we have seen earlier, except that the pivot is chosen cleverly in a recursive fashion. More precisely, line 4 of *Select* is changed to

```
4.   else Partition $S$ into $\lceil n/5 \rceil$ subsets $S_1, \dots, S_{\lceil n/5 \rceil}$, each of size at most 5.
          Find the median $m_i$ of each $S_i$. (Since $|S_i| \leqslant 5$ this takes $O(1)$ for each $i$.)
          Let $M = \{m_1, \dots, m_{\lceil n/5 \rceil}\}$.
          $x_{\text{piv}} \leftarrow$ Select($M$, $\lfloor (|M|+1)/2 \rfloor$)
```

Clearly the algorithm is still correct. Indeed, the only thing that changed is the pivot selection and the correctness does not depend on the choice of the pivot. What about the running time? We now have at most two recursive calls: one in line 4 on a set of $\lceil n/5 \rceil$ elements, and possibly one in line 10 or 11. On how many elements do we have to recurse in the worst case in this second recursive call? To bound this number, we must determine the rank of the pivot element x_{piv}. Note that there are (roughly) $|M|/2 = (n/5)/2 = n/10$ elements $m_i \in M$ that are smaller than x_{piv}. For each m_i, there are two more elements in S_i smaller than it. Hence, in total there are at least $3n/10$ elements smaller than x_{piv}. A similar argument shows that there are at least $3n/10$ elements larger than x_{piv}. Hence, both $S_<$ and $S_>$ contain at most $7n/10$ elements. Besides the at most two recursive calls, the algorithm needs $O(n)$ time. We thus get the following recurrence on $T(n)$, the worst-case running time of our algorithm:

$$T(n) = O(n) + T(n/5) + T(7n/10),$$

which solves to $T(n) = O(n)$.

2.2 Randomized QuickSort

The randomized version of the well known QuickSort algorithm can be described as follows.

```
Algorithm Randomized-QuickSort($S$)
1.  ▷ sorts set $S$ in increasing order
2.  if $|S| \leqslant 1$
3.     then return $S$
4.     else Pick a pivot element $x_{\text{piv}}$ from $S$ uniformly at random.
5.          $S_< \leftarrow \{x \in S : x < x_{\text{piv}}\}$;  $S_= \leftarrow \{x \in S : x = x_{\text{piv}}\}$;  $S_> \leftarrow \{x \in S : x > x_{\text{piv}}\}$.
6.          Randomized-QuickSort($S_<$);  Randomized-QuickSort($S_>$)
7.          return the sorted set obtained by concatenating $S_<$, $S_=$, and $S_>$ (in that order)
```

We have swept a few details under the rug in the description, in particular how to represent S.

One possibility is to store the elements in a linked list. Picking a random pivot element would then proceed as follows: set $r \leftarrow \textit{Random}(1, |S|)$, walk along S for r steps, and take the element that is reached as pivot. Generating linked lists for $S_<$, $S_=$, and $S_>$ is now easy: just walk along the list of S and put each elements in the list where it belongs. Concatenating the lists in line 7 is also easy.

Another possibility, which will probably be slightly more efficient in practice, will be to store S in an array $A[1..n]$. To select the pivot we set $r \leftarrow \textit{Random}(1, |S|)$ and then take $x_{\text{piv}} \leftarrow A[r]$. Next we re-arrange the elements in A such that the first part of A stores all elements smaller than x_{piv}, the second part all element equal to x_{piv}, and the last part all elements greater than x_{piv}. We can then recurse on the relevant parts of the array (the first and the last part), and we are done—the sorted subarrays are automatically in the right order so line 7 can be skipped. The details of this implementation are described in Chapter 7 of [CLRS].

In any case, the running time of the algorithm is $O(n)$ plus the time needed to recursively sort $S_<$ and $S_>$. If we are very unlucky, then the pivot is the smallest (or largest) element in S. This leads to the following recurrence for the worst-case running time, $T_{\text{worst}}(n)$:

$$T_{\text{worst}}(n) = O(n) + T_{\text{worst}}(n-1),$$

which solves to $T_{\text{worst}}(n) = O(n^2)$. On the other hand, if we are extremely lucky the random pivot would be the median every time, leading to a best-case running time satisfying

$$T_{\text{best}}(n) = O(n) + 2T_{\text{best}}(n/2).$$

This gives $T_{\text{best}}(n) = O(n \log n)$. But what about the *expected* running time?

We first observe that to construct the sets $S_<$, $S_=$, and $S_>$ in line 5 of the algorithm, we compare each element in S to the pivot element x_{piv}. Thus the time needed by the algorithm, not counting the time needed by the recursive calls, is O(number of comparisons). This even holds for the whole algorithm, including all recursive calls: the total time needed is

$$O(\text{ total number of comparisons made over all recursive calls }).$$

It remains to bound the expected number of comparisons made. To this end we introduce indicator random variables, X_{ij}. For the analysis it is convenient to denote the elements in

S by $x_1, \ldots, x_n$ in order.[1] Thus x_1 is the smallest element in S, and so on. Now define

$$X_{ij} := \begin{cases} 1 & \text{if } x_i \text{ and } x_j \text{ are compared during one of the recursive calls} \\ 0 & \text{otherwise} \end{cases}$$

Because any two elements are compared at most once, we have

$$\text{total number of comparisons made over all recursive calls } = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

To bound the expected running time, we must thus bound $\mathrm{E}[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}]$. We have

$$\begin{aligned} \mathrm{E}[\textstyle\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}] &= \textstyle\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] && \text{(linearity of expectation)} \\ &= \textstyle\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[\ x_i \text{ and } x_j \text{ are compared }] && \text{(definition of } X_{ij}\text{)} \end{aligned}$$

What is the probability that elements x_i and x_j (with $i < j$) are compared in one of the recursive calls? To answer this question, we consider what happens to the set $S_{ij} := \{x_i, x_{i+1}, \ldots, x_j\}$ during the algorithm. In the first call of the algorithm, for instance, we may choose a pivot x_{piv} that is smaller than all elements in S_{ij}. When this happens all elements will go into $S_>$. More generally, if the pivot is not an element of S_{ij} then S_{ij} will stay together in the sense that all elements go to the same recursive call. This continues until at some point one of the elements in S_{ij} is chosen as the pivot. Now there are two possibilities: either x_i or x_j is chosen as the pivot, or one of the elements $x_{i+1}, \ldots, x_{j-1}$ is chosen as the pivot. In the first case x_i and x_j are compared during this call. In the second case they are not compared in this call—both are only compared to the pivot. Moreover, in the second case they will not be compared in some later stage either, because x_i will go into $S_<$ and x_j will go into $S_>$. We therefore conclude that x_i and x_j are compared if and only if one of them is first chosen as a pivot among all elements in the set S_{ij}. Since each element of S_{ij} has equal probability of being chosen as pivot, we therefore find

$$\Pr[\ x_i \text{ and } x_j \text{ are compared }] = \frac{2}{j-i+1}.$$

Putting it all together, we get

$$\begin{aligned} \mathrm{E}[\textstyle\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}] &= \textstyle\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[\ x_i \text{ and } x_j \text{ are compared }] \\ &= \textstyle\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= 2\textstyle\sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k+1} \\ &< 2\textstyle\sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k} \\ &< 2\textstyle\sum_{i=1}^{n-1} O(\log(n-i)) \\ &= O(n \log n). \end{aligned}$$

We conclude that the expected running time *Randomized-QuickSort* is $O(n \log n)$.

[1] In the analysis we assume for simplicity that all elements are distinct. You should verify that in fact the analysis still holds when this is not the case.

An alternative approach. Instead of picking the pivot uniformly at random from S, we could also insist in picking a good pivot. An easy way to do this is to use algorithm *ApproxMedian3* to find a (1/4)-approximate median. Now the expected running time is bounded by

$$\text{E}[\text{ (running time of } \textit{ApproxMedian3} \text{ with } \delta = 1/4) + \text{(time for recursive calls) }].$$

Using linearity of expectation and that the expected running time of *ApproxMedian3* is $O(n/\delta)$, we see that the expected running time is

$$O(n) + \text{E}[\text{ time for recursive calls }].$$

Because the pivot is a (1/4)-approximate median, the most unbalanced partitioning that may result is one where $|S_<| = n/4$ and $|S_>| = 3n/4$ (or vice versa). Hence, the expected running time T_{exp} satisfies

$$T_{\text{exp}}(n) = O(n) + T_{\text{exp}}(3n/4) + T_{\text{exp}}(n/4),$$

which solves to $T_{\text{exp}}(n) = O(n \log n)$. Although the expected running time is also $O(n \log n)$, this algorithm will be slightly less efficient than *Randomized-QuickSort*. Indeed, when *ApproxMedian3* checks a random element x^* to see if it is a (1/4)-approximate median, it tests for all other elements whether they are smaller than x^*. Hence, it effectively computes the sets $S_<$ and $S_>$ and it may as well recursively sort those sets rather than throwing away the work done and starting from scratch.

Chapter 3

Randomized search structures

Let S be a set of elements, where each element x has a key $key[x]$ associated to it. A *dictionary* on a set S is an abstract data structure that supports the following operations:

- *Search*(S, k): return an element $x \in S$ such that $key[x] = k$ if such an element exists, and **nil** otherwise.
- *Insert*(S, x): insert element x with associated key $key[x]$ into the structure (thereby setting $S \leftarrow S \cup \{x\}$), where it is assumed that x is not yet present.
- *Delete*(S, x): delete element x from the structure (thereby setting $S \leftarrow S \setminus \{x\}$). Here the parameter of the procedure is (a pointer to) the element x, so we do not have to search for it.

If the set S is static—the operations *Insert* and *Delete* need not be supported—then a sorted array is a good structure: it is simple and a *Search* operation takes only $O(\log n)$ time, where $n = |S|$. Insertions into and deletions from a sorted array are expensive, however, so when S is dynamic then an array is not a good choice to implement a dictionary. Instead we can use a red-black tree or any other balanced search tree, so that not only *Search* but also *Insert* and *Delete* can be done in $O(\log n)$ time in the worst case—see Chapters 12 and 13 from [CLRS] for more information on search trees and red-black trees. Even though red-black trees are not very difficult to implement, the details of *Insert* and *Delete* are somewhat tedious. Here we shall look at two simpler, randomized solutions. For simplicity of presentation we will assume that all the keys are distinct.

3.1 Treaps

A binary search tree on a S is a binary tree whose internal nodes store the elements from S such that the *search-tree property* holds:

> Let ν be a node in the tree storing an element x. Then for any element y stored in the left subtree of ν we have $key[y] < key[x]$, and for any element z in the right subtree of ν we have $key[x] < key[z]$.

A search operation on a binary search tree takes time linear in the depth of the tree. Hence, we want a tree with small depth. In a so-called *treap* this is achieved as follows.

We also assign a priority $prio[x]$ to each element $x \in S$, where we assume that all priorities are distinct. A treap on S is a binary tree storing the element from S in its internal nodes such that it has the search-tree property with respect to its keys and the *heap property* with respect to its priorities:

> Let ν be a node in the heap storing an element x. Then for any element y stored in the left or right subtree of ν we have $prio[y] < prio[x]$.

Is it always possible to satisfy both the search-tree property and the heap property? The answer is yes. However, for any set S with given keys and priorities there is only one way to do this.

Lemma 3.1 *There is a unique treap for any set S.*

Proof. By induction on n, the number of elements in S. (Exercise.) □

The idea of a treap is to assign each element in S a random priority. To avoid some technical difficulties we shall assume that we can generate a real number uniformly at random in the interval $[0, 1]$ in $O(1)$ time. If we now choose $prio[x]$ uniformly at random from $[0, 1]$ for each $x \in S$ independently, then all priorities are distinct with probability one.[1]

The hope is of course that by assigning random priorities, the treap is expected to be rather well balanced. The next lemma shows that this is indeed the case. Define the *depth* of an element x to be the number of nodes on the path from the root of the treap to x.[2] (For example, the depth of the root is 1, the depth of its children is 2, etc.) Let's number the elements from S as $x_1, \ldots, x_n$ according to their rank, so $rank(x_i) = i$.

Lemma 3.2

$$\mathrm{E}[depth(x_k)] = H_k + H_{n-k+1} - 1,$$

where $H_j := \sum_{i=1}^{j} \frac{1}{i} \approx \ln j$.

Proof. We want to count the expected number of elements x_i that are an ancestor of—that is, that are on the path to—the node storing x_k. To this end we introduce indicator random variables

$$X_i := \begin{cases} 1 & \text{if } x_i \text{ is an ancestor of } x_k \\ 0 & \text{otherwise} \end{cases}$$

Thus we want to bound $\mathrm{E}[\sum_{i=1}^{n} X_i]$. By linearity of expectation, this boils down to bounding the values $\mathrm{E}[X_i]$. By definition of X_i we have $\mathrm{E}[X_i] = \Pr[\ x_i \text{ is an ancestor of } x_k\ ]$. What is this probability? Suppose first that $i \leqslant k$. We claim that x_i is an ancestor of x_k if and only if $prio(x_i) > prio(x_j)$ for all $i < j \leqslant k$. In other words, if and only if x_i has the highest priority among the elements in $\{x_i, x_{i+1}, \ldots, x_k\}$. Before we prove this claim, we show that it implies the lemma. Indeed, since the priorities are chosen independently and uniformly at random from $[0, 1]$, each of the elements in $\{x_i, x_{i+1}, \ldots, x_k\}$ has the same probability of getting the

[1]Of course we cannot really generate random real numbers. Instead we can generate enough random bits from the binary representation of each priority to ensure all priorities are distinct. To focus on the main ideas behind treaps and their analysis, we will ignore this issue and just assume we can generate random real numbers.

[2]In fact, we should say "to the node of the tree where x is stored." To simplify the presentation we will permit ourselves this slight abuse of terminology and we will not distinguish between the elements and the nodes where they are stored.

highest priority. This probability is simply $1/(k-i+1)$. Similarly, for an element x_i with $i \geqslant k$, we have $\Pr[\ x_i \text{ is an ancestor of } x_k\ ] = 1/(i-k+1)$. We get

$$\begin{aligned}
\mathrm{E}[\textstyle\sum_{i=1}^{n} X_i] &= \textstyle\sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \textstyle\sum_{i=1}^{n} \Pr[\ x_i \text{ is an ancestor of } x_k\ ] \\
&= \textstyle\sum_{i=1}^{k} \Pr[\ x_i \text{ is an ancestor of } x_k\ ] + \sum_{i=k}^{n} \Pr[\ x_i \text{ is an ancestor of } x_k\ ] - 1 \\
&= \textstyle\sum_{i=1}^{k} \frac{1}{k-i+1} + \sum_{i=k}^{n} \frac{1}{i-k+1} - 1 \\
&= \textstyle\sum_{i=1}^{k} \frac{1}{i} + \sum_{i=1}^{n-k+1} \frac{1}{i} - 1 \\
&= H_k + H_{n-k+1} - 1
\end{aligned}$$

It remains to prove our claim that x_i is an ancestor of x_k (for $i < k$) if and only if $\mathit{prio}(x_i) > \mathit{prio}(x_j)$ for all $i < j \leqslant k$.

To prove the if-part of the statement, assume that $\mathit{prio}(x_i) > \mathit{prio}(x_j)$ for all $i < j \leqslant k$. By the heap property, x_i cannot be in the subtree rooted at x_k. Moreover, there cannot be an ancestor x_j of x_k with x_i and x_k in different subtrees. Such an element would have higher priority than x_i and would satisfy $x_i < x_j < x_k$, contradicting our assumption that $\mathit{prio}(x_i) > \mathit{prio}(x_j)$ for all $i < j \leqslant k$. Hence, x_i must be an ancestor of x_k if $\mathit{prio}(x_i) > \mathit{prio}(x_j)$ for all $i < j \leqslant k$.

To prove the only-if-part, take an ancestor x_i of x_k and suppose for a contradiction that there is some x_j with $i < j \leqslant k$ that has higher priority than x_i. By the heap property, x_j cannot be stored in the subtree rooted at x_i. But x_j cannot be stored at an ancestor of x_i either: we have $x_i < x_j \leqslant x_k$, so x_i would be stored in the left subtree of x_j and x_k would not be stored in that subtree, contradicting that x_i is an ancestor of x_k. The only remaining possibility is that there is an ancestor x_ℓ of x_i that has x_i in one of its subtrees and x_j in the other. But together with $x_i < x_j \leqslant x_k$ this again contradicts that x_i is an ancestor of x_k, since x_k would be in the same subtree of x_ℓ as x_j. We conclude that if x_i is an ancestor of x_k then $\mathit{prio}(x_i) > \mathit{prio}(x_j)$ for all $i < j \leqslant k$. $\square$

We have proved that the expected depth of any element $x \in S$ is $O(\log n)$, which means that the expected search time for any element is $O(\log n)$. Can we conclude that the expected depth of the whole tree $\mathcal{T}$ is $O(\log n)$? Not immediately. The problem is that in general for a collection $Y_1, \ldots, Y_n$ of random variables $\mathrm{E}[\max_i Y_i] \neq \max_i \mathrm{E}[Y_i]$: the expected maximum of a collection of random variables is not necessarily the same as the maximum of the expectations of those random variables. In particular

$$\mathrm{E}[\max_{x \in S} \mathit{depth}(x)] \neq \max_{x \in S} \mathrm{E}[\mathit{depth}(x)].$$

Stated differently, the fact that the depth of every *individual* $x \in S$ is expected to be $O(\log n)$ does not imply that the depths of all of them are expected to be $O(\log n)$ *simultaneously.* For this we need a stronger statement about the individual depth: instead of just saying that the expected depth of any $x \in S$ is $O(\log n)$, we must argue that the depth of every x is $O(\log n)$ *with high probability*, that is, that the probability that the depth is $O(\log n)$ is of the form $1 - \frac{1}{n^c}$ from some constant $c > 0$.

Lemma 3.3

$$\Pr[\ \mathit{depth}(x_k) \leqslant 7(H_k + H_{n-k+1} - 1)\ ] \ \geqslant\ 1 - (1/n^6).$$

Proof. Recall from the proof of Lemma 3.2 that the depth of an element x_k can be written as $\sum_{i=1}^{n} X_i$, where X_i is the indicator random variable for the event "x_i is an ancestor of x_k". We now observe that the events "x_i is an ancestor of x_k" and "x_j is an ancestor of x_k" are independent for $i \neq j$. Indeed, x_i is an ancestor of x_k (for some $i < k$) if and only if x_i has the highest priority among $\{x_i, \ldots, x_k\}$, and the probability of this happening is not influenced by the fact that x_j has the highest priority in some subset $\{x_j, \ldots, x_k\}$. Hence, we are in the situation of n independent Poisson trials $X_1, \ldots, X_n$. This means we can use the following result: for Poisson trials, if we let $X = \sum_{i=1}^{n} X_i$ and $\Pr[X_i = 1] = p_i$, then we have for any $\delta > 0$ that

$$\Pr[X > (1+\delta)\mu] \leqslant \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu},$$

where $\mu = \mathrm{E}[X]$ and $e = 2.718\ldots$ (the base of the natural logarithm). Applying this result with $\delta = e^2 - 1 > 6$ and noting that $H_k + H_{n-k+1} - 1 \geqslant H_n$ for all k, we get

$$\begin{array}{rcll} \Pr[\mathit{depth}(x_k) > (1+\delta)(H_k + H_{n-k+1} - 1)] & \leqslant & \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{H_n} & \\ & \leqslant & \left(\frac{e^{\delta}}{(e^2)^{1+\delta}}\right)^{H_n} & \\ & \leqslant & \left(\frac{1}{e}\right)^{\delta H_n} & \\ & \leqslant & \left(\frac{1}{n}\right)^{6} & \text{(since } H_n \geqslant \ln n \text{ and } \delta > 6) \end{array}$$

□

Okay, we have established that every $x_i \in S$ has depth $O(\log n)$ with high probability, namely probability at least $1 - (1/n)^6$. What is the probability that all elements have depth $O(\log n)$ simultaneously? To bound this probability we define A_i as the event "$\mathit{depth}(x_i) \geqslant 7(H_k + H_{n-k+1} - 1)$". Now we use that the probability that at least one of a given set of events takes place is upper bounded by the sum of the probabilities of the individual events. Hence

$$\Pr[A_1 \cup \cdots \cup A_n] \;\leqslant\; \Pr[A_1] + \ldots + \Pr[A_n] \;\leqslant\; n \cdot (1/n^6) \;=\; 1/n^5.$$

We conclude that the depth of the whole tree is $O(\log n)$ with high probability. (Note that since the depth is never more than n, this also implies that the expected depth is $O(\log n)$.)

We have seen that the search time in a treap is $O(\log n)$ with high probability. Now what about insertions and deletions into a treap?

First consider the insertion of a new element x with key $\mathit{key}[x]$ into the treap $\mathcal{T}$. We do this as follows. We assign x a priority by taking a real number in the range $[0, 1]$ uniformly at random. Then we first insert x into $\mathcal{T}$ without paying attention to its priority: we just search in $\mathcal{T}$ with the value $\mathit{key}[x]$ to replace the leaf where the search ends with a node storing x. This is exactly as one would insert into an unbalanced search tree—see Chapter 12 from [CLRS]. After doing this, the new tree $\mathcal{T}$ will have the search-tree property. Of course it may happen that the priority of x is higher than the priority of its parent, thus violating the heap property. When this happens we perform a rotation around the parent of x—see Chapter 13 of [CLRS] for a description of rotations—thus bringing x one level higher up in the tree. If the priority of x is still higher than the priority of its parent, we move x up one more level by doing another rotation. This continues until the priority of x is lower than that of its parent (or x becomes the root), at which moment the heap property is restored. Since at each rotation x moves up a level in $\mathcal{T}$, we do at most $depth(\mathcal{T})$ rotations, and the insertion runs in time $O(\log n)$ with high probability.

Deletions work in the reverse way: first we move x down in the tree by rotations (pretending it has priority $-\infty$) until both its children are leaves, and then we can easily delete x.

3.2 Skip lists

Let S be a set of n elements. Let's slightly generalize the search operation, so that when there is no element x with $key[x] = k$ it returns the element $x \in S$ with the largest key smaller than k. We call this element the *predecessor* of k, and we call the query a *predecessor query*. Thus $Predecessor(S, k)$ returns the element with the largest key smaller than or equal to k.

A skip list for a set S consists of a number of sorted linked lists $\mathcal{L}_0, \mathcal{L}_1, \dots, \mathcal{L}_h$. Each list $\mathcal{L}_i$ stores a subset $S_i \subset S$, such that $S_0 = S$, and $S_i \subset S_{i-1}$ for all $0 < i \leqslant h$, and $S_h = \emptyset$. Each sorted list also stores two dummy elements, one with key $-\infty$ at the beginning of the list and one with key $+\infty$ at the end of the list. For a set S_i (or list $\mathcal{L}_i$) we call i the *level* of that set (or list), and we call h the *height* of the skip list. We also have pointers between consecutive lists. More precisely, for every element $x \in S_i$ (with $i > 0$) we have a pointer from its occurrence in $\mathcal{L}_i$ to its occurrence in $\mathcal{L}_{i-1}$—see Fig. 3.1 for an example.

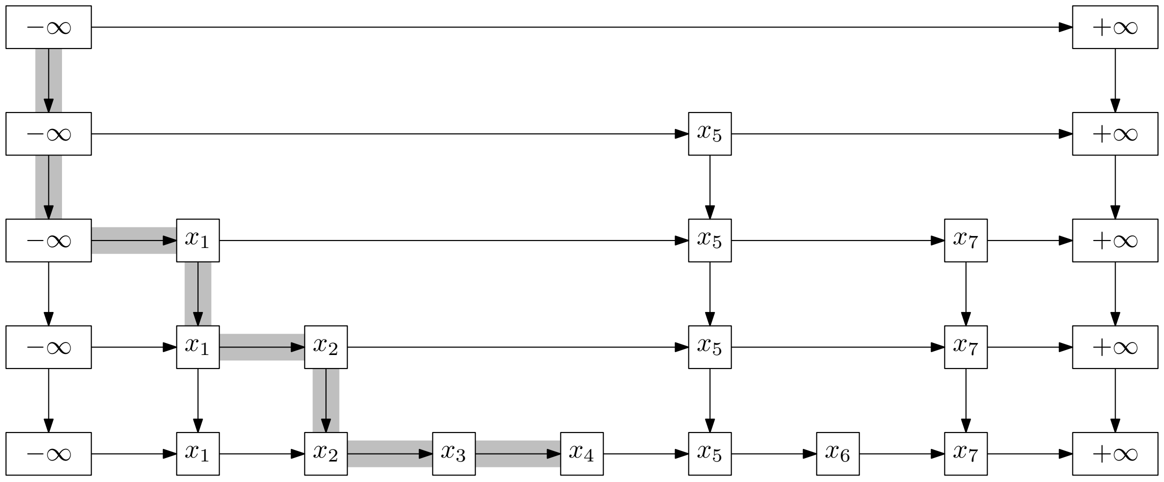

Figure 3.1: A skip list on a set of seven elements. The search path taken by a predecessor query with query value k (for some k with $key[x_4] \leqslant k < key[x_5]$) is indicated.

Next we describe the algorithm for performing a predecessor query in a skip list $\mathcal{S}$. We denote the pointer from an element $x \in \mathcal{L}_i$ to the next element in $\mathcal{L}_i$ by $next[x]$, and the pointer from x in $\mathcal{L}_i$ to the copy of x in $\mathcal{L}_{i-1}$ by $down[x]$.

```
Algorithm Predecessor($\mathcal{S}, k$)
1.  Let $y$ be the dummy element $-\infty$ of $\mathcal{L}_h$.
2.  for $i \leftarrow h$ downto 1
3.      do $y \leftarrow down[y]$
4.          while $key[next[y]] \leqslant k$
5.              do $y \leftarrow next[y]$
6.  return $y$
```

Recall that S_0 is the whole set S. The main question is how the sets S_i, $i \geqslant 1$, should be chosen. Let $x_1, x_2, \ldots, x_n$ denote the elements from S in sorted order. One idea would be to take $S_1 = \{x_2, x_4, x_6, \ldots\}$, to take $S_2 = \{x_4, x_8, \ldots\}$, and so on. In other words, we obtain S_i by deleting the first, third, fifth, etc. element from S_{i-1}. This way the depth of the skip list would be $O(\log n)$ and the **while** -loop of lines 4–5 of *Predecessor* would do at most one step step forward in the list $\mathcal{L}_i$. However, this would make updating the skip list very costly. Hence, skip lists employ the following simple but powerful idea: for each element $x \in S_{i-1}$ we flip a fair coin; if the outcome is HEADS we put x into S_i, if the outcome is TAILS then we do not put x into S_i. In other words, each element in S_{i-1} is selected independently with probability 1/2. Since each element is selected with probability 1/2, we expect that $|S_i| = |S_{i-1}|/2$. In other words, the number of elements is expected to halve at every level. The following lemma proves that the height is $O(\log n)$ *with high probability*, a result that we will need to prove a bound on the query time.

Lemma 3.4 *The height of a skip list on a set of n elements is more than $1 + t \log n$ with probability less than $1/n^{t-1}$.*

Proof. Consider an element $x \in S$. We define the *height* of x to be the largest i such that $x \in S_i$. Thus the height of the skip list is equal to $1 + \max_{x \in S} \mathit{height}(x)$. Now observe that for an element x to have height at least s, it must have had s successive coin flips turn up HEADS. Hence,

$$\Pr[\ \mathit{height}(x) \geqslant s\ ] = (1/2)^s.$$

Setting $s = t \log n$ we get

$$\begin{aligned}
&\Pr[\ \max_{x \in S} \mathit{height}(x) \geqslant t \log n\ ] \\
&\quad = \ \Pr[\ (\mathit{height}(x_1) \geqslant t \log n) \vee \cdots \vee (\mathit{height}(x_n) \geqslant t \log n)\ ] \\
&\quad \leqslant \ \Pr[\ (\mathit{height}(x_1) \geqslant t \log n)\ ]\ +\ \cdots\ +\ \Pr[\ (\mathit{height}(x_n) \geqslant t \log n)\ ] \\
&\quad \leqslant \ n/2^{t \log n} \\
&\quad = \ 1/n^{t-1}.
\end{aligned}$$

□

We are now ready to prove a bound on the query time in a skip list.

Lemma 3.5 *The expected time to answer a predecessor query in a skip list is $O(\log n)$.*

Proof. The query time is equal to

$$O(\sum_{i=0}^{h} (1 + \ \#\ \textit{next}\text{-pointers followed in } \mathcal{L}_i).$$

Let X_i be a random variable denoting the number of *next*-pointers followed in $\mathcal{L}_i$. We want to bound

$$\mathrm{E}[\sum_{i=0}^{h} (1 + X_i)].$$

Recall that S_{i+1} is obtained from S_i by sampling every element independently with probability 1/2. How many *next*-pointers do we have to follow in $\mathcal{L}_{i+1}$? Well, we start at an element x_j

in S_i that is also present in S_{i+1} and then walk to the right. We cannot reach an element x_ℓ of S_i that is also present in S_{i+1}, otherwise we would already have reached x_ℓ in $\mathcal{L}_{i+1}$. Hence, the number of *next*-pointers followed in $\mathcal{L}_i$ is at most the number of elements in S_i between x_j and x_ℓ, the smallest element greater than x_j in S_{i+1}. Since the elements x_{j+1}, x_{j+2}, etc. are all present in S_{i+1} with probability 1/2, we expect that in between two successive elements in S_{i+1} there is only one element in S_i. Hence, $\mathrm{E}[X_i] \leqslant 1$. Using linearity of expectation we obtain a bound on the number of *next*-pointers followed in the first $3 \log n$ levels:

$$\mathrm{E}[\sum_{i=0}^{3\log n}(1+X_i)] = O(\log n) + \sum_{i=0}^{3\log n}\mathrm{E}[X_i] = O(\log n).$$

For $i > 3\log n$ we of course also have $\mathrm{E}[X_i] \leqslant 1$. However, the total number of levels could be much more than logarithmic, so we have to be a bit careful. Fortunately Lemma 3.4 tells us that the probability that the total number of levels is more than $1 + 3\log n$ is quite small. Combining this with the fact that obviously $X_i \leqslant n$ and working out the details, we get:

$$\begin{aligned}
\mathrm{E}[\textstyle\sum_{i\geqslant 3\log n}(1+X_i)] &\leqslant \textstyle\sum_{i\geqslant 3\log n}\Pr[\text{ height of skip list} \geqslant i\ ]\cdot(1+n)\\
&\leqslant \textstyle\sum_{t\geqslant 3}\Pr[\text{ height of skip list} \geqslant t\log n\ ]\cdot(1+n)\log n\\
&\leqslant \textstyle\sum_{t\geqslant 3}(1/n^{t-1})\cdot(1+n)\log n\\
&= O(1).
\end{aligned}$$

□

It remains to describe how to insert and delete elements. We only sketch these operations, leaving the details as an exercise.

Suppose we want to insert an element x. The first idea that comes to mind is to perform a predecessor query with *key*$[x]$ to find the position of x in the bottom list $\mathcal{L}_0$. Then we throw a coin to decide if x also has to be inserted into $\mathcal{L}_1$. If so, we insert x into $\mathcal{L}_1$ and throw a coin to see if we also need to insert x into $\mathcal{L}_2$, and so on. The problem with this approach is that a skip list does not have up-pointers, so we cannot go from $\mathcal{L}_0$ to $\mathcal{L}_1$ for instance. Hence, we proceed differently, as follows. First, we determine *height*(x), the number of lists into which we need to insert x. To this end we flip a coin until we get TAILS; the number of coin flips then gives us *height*(x). Then we search the skip list with *key*$[x]$, inserting x into the relevant lists. (Note that it can happen that *height*(x) is larger than the current height of the entire skip list. When this happens the skip list will get a new level above the already existing levels.)

To delete an element x from a skip list we walk down the skip list, making sure that at every level we reach the largest element smaller than *key*$[x]$. In the lists $\mathcal{L}_i$ containing x, this will give us the element just before x, so we can easily delete x from those lists.

Chapter 4

Randomized permutation routing on the hypercube

In this chapter we study a beautiful example of the power of randomization in parallel computing. Consider a parallel machine with n processors, labeled $0, \ldots, n-1$. The processors are linked together in a network, so that they can communicate. However, not every pair of processors has a link between them, which means that some processors can only communicate via other processors. A fairly standard topology for such a network of processors is the *hypercube*. Here the number of processors, n, is a power of 2 and there is a (bidirectional) link between processors i and j if and only if the binary representation of i and j differ in exactly one bit. For example, a hypercube with $n = 8$ processors will have the following links:

$$(0,1),(0,2),(0,4),(1,3),(1,5),(2,3),(2,6),(3,7),(4,5),(4,6),(5,7),(6,7).$$

The link $(1,3)$ exists, for instance, because 001 (=1) and 011 (=3) differ only in the second bit. From now on we will consider the two opposite links of a bidirectional link as separate links. (So for $n = 8$ we would, besides the links above, also have the links $(1,0)$, $(2,0)$, etc.)

Now suppose each processor i, for $1 \leqslant i \leqslant n$, wants to send a message $\mathit{msg}(i)$ to some other processor $\mathit{dest}(i)$. We consider the case of *permutation routing*, where the set $\{\mathit{dest}(1), \ldots, \mathit{dest}(n)\}$ of destinations forms a permutation of $1, \ldots, n$. In other words, each processor i wants to send exactly one message and wants to receive exactly one message. Sending these messages is done in *rounds*. In each round, a processor can send at most one message to each of its neighbors. Each processor i has a collection of $\log n$ buffers, one for each outgoing link. We denote the buffer for the link (i,j) by $\mathcal{B}_i(j)$. Buffer $\mathcal{B}_i(j)$ will store messages that processor i needs to forward to its neighbor j, but that have to wait because i also needs to forward other messages to j. Each processor i executes the following algorithm in every round:

```
Algorithm RoutingStrategy($i$)
1.  ▷ Send phase:
2.      for each outgoing link $(i,j)$
3.          do Select a message from $\mathcal{B}_i(j)$ and send it along link $(i,j)$.
4.      ▷ Receive phase:
5.      for each incoming message whose destination is not $i$
6.          do Store that message in $\mathcal{B}_i(j)$, where $j$ is the next processor on its route.
```

The main question now is how the routes are chosen. The goal is to do this in such a way that the total number of rounds needed before every message has arrived at its destination is as small as possible. Thus we would like the routes to be short. Moreover, we do not want too many routes to use the same links in the network because that will lead to congestion: the buffers of these congested links will have to store many messages, and consequently these messages will have to wait a long time before they are finally forwarded. This raises another issue that we still have to address: how does a processor i select from its buffers $\mathcal{B}_i(j)$ which messages to send in each round? Note that a processor i does not necessarily know the destinations $\mathit{dest}(j)$ of the messages of the other processors. Hence, the route-planning algorithm should be *oblivious*: each processor should determine $\mathit{route}(\mathit{msg}(i))$ based only on i and $\mathit{dest}(i)$, not on any $\mathit{dest}(j)$ for $j \neq i$.

A simple routing strategy is the following:

- Each buffer $\mathcal{B}_i(j)$ is implemented as a queue, and the incoming messages in each round are put into the queue in arbitrary order. (Note: *arbitrary* means that the order in which the messages are put in the queue does not matter, it does *not* mean that we put them into the queue in random order.)
- The route followed by each message is determined using the *bit-fixing strategy*, which is defined as follows. Suppose we have already constructed some initial portion of $\mathit{route}(\mathit{msg}(i))$ and let j be the last processor on this initial portion. To determine the next processor j' on the route we look at the bits in the binary representation of $\mathit{dest}(i)$ from left to right, and determine the leftmost bit b that is different from the corresponding bit of the binary representation of j. We then take j' such that its binary representation is the same as that of j, except for the bit b. (Note that because we are routing on a hypercube, the link (j, j') exists.) For example, if $n = 32$ and processor 01101 wants to send a message to 00110, then the route would be

$$01101 \rightarrow 00101 \rightarrow 00111 \rightarrow 00110.$$

This routing strategy is pretty simple and it is oblivious. Moreover, it has the advantage that a processor does not need to include the whole route for $\mathit{msg}(i)$ into the message header, it suffices to put $\mathit{dest}(i)$ into the header. (Based on $\mathit{dest}(i)$ and its own processor number j, the processor j knows where to send the message $\mathit{msg}(i)$ to.) Unfortunately, the number of rounds can be fairly large: there are sets of destinations on which this routing strategy needs $\Omega(\sqrt{n})$ rounds. In fact, one can prove that for *any* deterministic routing strategy there is a set of destinations that will require $\Omega(\sqrt{n/\log n})$ rounds. Surprisingly, using randomization one can do much better. The trick is to let each processor i first route its message to a random intermediate destination:

```
Algorithm DetermineRoute(i)
1.  $r_i \leftarrow Random(0, n-1)$
2.  Construct a route from processor $i$ to processor $r_i$ using the bit-fixing strategy,
    and construct a route from processor $r_i$ to processor $dest(i)$ using the bit-fixing
    strategy. Let $route(msg(i))$ be the concatenation of these two routes.
```

Next we analyze the expected number of rounds this randomized routing algorithm needs. For simplicity we will slightly change the algorithm so that it consists of two phases: in the first phase all messages $msg(i)$ will be routed from i to r_i, and in the second phase all messages will be routed from r_i to $dest(i)$. Thus the messages wait at their intermediate destination until all messages have arrived at their intermediate destination. We will analyze the number of rounds needed in the first phase. By symmetry the same analysis holds for the second phase. From now on, we let $route(i)$ denote the route from i to r_i, as prescribed by the bit-fixing strategy. Our analysis will be based on the following lemma, which we will not prove:

Lemma 4.1 *Let S_i be the set of messages whose route uses at least one of the links in $route(i)$. Then the delay incurred by $msg(i)$ in phase 1 is at most $|S_i|$. In other words, $msg(i)$ reaches its intermediate destination r_i after at most $|route(i)| + |S_i|$ rounds, where $|route(i)|$ is the length of the path $route(i)$.*

To analyze the algorithm, we introduce indicator random variables X_{ij}:

$$X_{ij} := \begin{cases} 1 & \text{if } j \neq i \text{ and } route(i) \text{ and } route(j) \text{ share at least one link} \\ 0 & \text{otherwise} \end{cases}$$

Now fix some processor i. By the previous lemma, $msg(i)$ will reach r_i after at most

$$|route(i)| + \sum_{0 \leqslant j < n} X_{ij}$$

steps.

First we observe that $|route(i)|$ is equal to the number of bits in the binary representation of i that are different from the corresponding bits in the representation of r_i. Hence, $\mathrm{E}[|route(i)|] = (\log n)/2$.

Next we want to bound $\mathrm{E}[\sum_{0 \leqslant j < n} X_{ij}]$. To this end we use the fact that the expected length of each route is $(\log n)/2$, as observed earlier. Hence, the expected total number of links used over all routes is $(n/2)\log n$. On the other hand, the total number of links in the hypercube is $n \log n$—recall that (i,j) and (j,i) are considered different links. By symmetry, all links in the hypercube have the same expected number of routes passing through them. Since we expect to use $(n/2)\log n$ links in total and the hypercube has $n \log n$ links, the expected number of routes passing through any link is therefore $\frac{(n/2)\log n}{n \log n} = 1/2$. This implies that if we look at a single link on $route(i)$ then we expect less than 1/2 other routes to use this link. Since $|route(i)| \leqslant \log n$, we get

$$\mathrm{E}[\sum_{0 \leqslant j < n} X_{ij}] \quad < \quad |route(i)|/2 \quad \leqslant \quad (\log n)/2.$$

This is good news: message $msg(i)$ is expected to arrive at its intermediate destination r_i within $\log n$ rounds. But we are not there yet. We want to achieve that *all* messages arrive at their destination quickly. To argue this, we need a high-probability bound on the delay of a message. To get such a bound we note that for fixed i, the random variables X_{ij} are independent. This is true because the intermediate destinations r_j are chosen independently. This means we are in the setting of independent Poisson trials: each X_{ij} is 1 or 0 with a certain probability, and the X_{ij} are independent. Hence we can use tail estimates for Poisson trials—see p.2 of the handouts on basic probability theory—to conclude that for $\delta > 0$

$$\Pr[X > (1+\delta)\mu] \leqslant \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}} \right)^{\mu},$$

where $X = \sum_{0 \leqslant j < n} X_{ij}$ and $\mu = \mathrm{E}[X]$. Plugging in $\delta = \frac{3 \log n}{\mu} - 1$ and using that $\mu \leqslant (\log n)/2$ (note that this implies $\delta \geqslant 5$) one can deduce that

$$\Pr[X > 3 \log n] \leqslant (1/2)^{2 \log n} = 1/n^2.$$

This implies that with probability at least $1 - 1/n$ *all* messages arrive with a delay of at most $3 \log n$ steps—much better than the $\Omega(\sqrt{n/\log n})$ lower bound for deterministic routing strategies.

Chapter 5

The probabilistic method

Randomization can not only be used to obtain simple and efficient algorithms, it can also be used in combinatorial proofs. This is called the *probabilistic method.* The general idea is to use one of the following two facts.

- Let X be a random variable. There is at least one elementary event for which the value X is less than or equal to $\mathrm{E}[X]$, and there is at least one elementary event for which the value of X is at least $\mathrm{E}[X]$.

- Consider a collection S of objects. If an object chosen at random from S has a certain property with probability greater than zero, then there must be an object in S with that property.

  Note: to prove the existence of an object in S with the desired property it is sufficient to prove that a random object in S has the property with positive probability. If you can even prove that a random object has the desired property with "large" probability, say at least some constant $p > 0$ that is independent of $|S|$, and you can check whether the object has the property, then you immediately obtain a Las Vegas algorithm for finding an object with the property.

We will consider two simple applications of the probabilistic method.

5.1 MaxSat

Let $x_1, \ldots, x_n$ be a set of n boolean variables. A boolean formula is a CNF formula—in other words, is in conjunctive normal form—if it has the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

where each clause C_j is the disjunction of a number of literals (a literal is a variable x_i or its negation $\overline{x_i}$). For example, the following is a CNF formula with three clauses.

$$(x_1 \vee x_3 \vee \overline{x_4}) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_5}) \wedge (x_1 \vee \overline{x_5}).$$

Given a CNF formula with m clauses, the MaxSat problem is to find a truth assignment satisfying as many clauses as possible.

Instead of looking at this algorithmic question, we can also consider the combinatorial question of how many clauses one can always satisfy. The following simple example shows there are CNF formulas where we cannot satisfy more than half the clauses:

$$(x_1) \wedge (\overline{x_1}) \wedge (x_2) \wedge (\overline{x_2}) \wedge \cdots \wedge (x_{m/2}) \wedge (\overline{x_{m/2}}).$$

Can we always satisfy at least half the clauses? And what if all the clauses have more than a single variable, can we perhaps satisfy more than $m/2$ clauses? The next theorem shows that this is indeed the case.

Theorem 5.1 *Any CNF formula with m clauses such that each clause has at least k variables has a truth assignment satisfying at least $(1-(1/2)^k)m$ clauses.*

Proof. Let $C_1 \wedge \cdots \wedge C_m$ be a CNF formula. Take a random truth assignment: For each variable x_i independently, flip a fair coin; if it comes up HEADS then we set $x_i =$ **true**, otherwise we set $x_i :=$ **false**. For each clause C_j, let X_j be the indicator random variable that is 1 if C_j is true and 0 otherwise. Then the number of satisfied clauses is $\sum_{j=1}^{m} X_j$. By linearity of expectation we have

$$\mathrm{E}[\sum_{j=1}^{m} X_j] = \sum_{j=1}^{m} \mathrm{E}[X_j] = \sum_{j=1}^{m} \Pr[\ C_j \text{ is satisfied }].$$

Now consider a clause C_j with $\ell \geqslant k$ literals. Since C_j is the disjunction of its literals, the only way in which C_j can be false is that all its literals are false. Since the truth values of the random variables are set randomly and independently, this happens with probability at most $(1/2)^\ell$. Hence,

$$\Pr[\ C_j \text{ is satisfied }] \geqslant 1-(1/2)^\ell \geqslant 1-(1/2)^k.$$

It follows that the expected number of satisfied clauses is at least $(1-(1/2)^k)m$. We conclude that there must be one truth assignment that satisfies at least this many clauses. $\square$

It follows for instance that any CNF formula has a truth assignment with at least $\lceil m/2 \rceil$ satisfied clauses. Also, any 3-CNF formula—any CNF formula where every clause has three literals—has a truth assignment where at least $7m/8$ clauses are satisfied.

5.2 Independent set

Let $G = (V, E)$ be a graph. A subset $V^* \subset V$ is called an *independent set* in G if no two vertices in V^* are connected by an edge in E.

Theorem 5.2 *Let $G = (V, E)$ be a graph with $|E| > |V|/2$. Then G has an independent set of size at least $\frac{|V|^2}{4|E|}$.*

Proof. Pick a random subset $V_R \subset V$ by taking each vertex with probability p independently, where p is a parameter to be determined later. Note that the expected number of vertices in V_R is $p|V|$. Let G_R be the subgraph induced by R, that is, $G_R = (V_R, E_R)$ where $E_R = \{(u, v) : u, v \in V_R \text{ and } (u, v) \in E\}$. An edge $(u, v) \in E$ is present in E_R with probability p^2. For an edge $e = (u, v) \in E$, define X_e as the indicator random variable that is 1 if $e \in E_r$

and 0 otherwise. Using linearity of expectation we can bound the expected number of edges in E_R by

$$\mathrm{E}[|E_R|] = \mathrm{E}[\sum_{e \in E} X_e] = \sum_{e \in E} \mathrm{E}[X_e] = p^2|E|.$$

One idea would be to choose $p < 1/\sqrt{|E|}$. Then the expected number of edges in E_R is less than 1, so that V_R would be an independent set with positive probability. This would give an independent set of size $p|V| < |V|/\sqrt{|E|}$. The statement of the theorem, however, promises a much larger independent set, namely $(1/4) \cdot (|V|/\sqrt{|E|})^2$. We therefore proceed differently. Instead of taking V_R itself as the independent set, we remove for each edge $e \in E_R$ one of its endpoints from V_R. Let V^* denote the resulting set of vertices. Clearly V^* is an independent set, and the size of V^* is at least

$$\mathrm{E}[|V_R| - |E_R|] = \mathrm{E}[|V_R|] - \mathrm{E}[|E_R|] \geqslant p|V| - p^2|E|.$$

Setting $p = |V|/(2|E|)$—note that $0 < p < 1$ because $|E| > |V|/2$—gives the desired bound. □

Chapter 6

Introduction to Approximation Algorithms

Many important computational problems are difficult to solve optimally. In fact, many of those problems are *NP-hard*[1], which means that no polynomial-time algorithm exists that solves the problem optimally unless P=NP. A well-known example is the *Euclidean traveling salesman problem (Euclidean TSP)*: given a set of points in the plane, find a shortest tour that visits all the points. Another famous NP-hard problem is *independent set*: given a graph $G = (V, E)$, find a maximum-size independent set $V^* \subset V$. (A subset is independent if no two vertices in the subset are connected by an edge.)

What can we do when faced with such difficult problems, for which we cannot expect to find polynomial-time algorithms? Unless the input size is really small, an algorithm with exponential running time is not useful. We therefore have to give up on the requirement that we always solve the problem optimally, and settle for a solution close to optimal. Ideally, we would like to have a guarantee on how close to optimal the solution is. For example, we would like to have an algorithm for Euclidean TSP that always produces a tour whose length is at most a factor ρ times the minimum length of a tour, for a (hopefully small) value of ρ. We call an algorithm producing a solution that is guaranteed to be within some factor of the optimum an *approximation algorithm.* This is in contrast to *heuristics*, which may produce good solutions but do not come with a guarantee on the quality of their solution.

Basic terminology. From now on we will use $\text{OPT}(I)$ to denote the value of an optimal solution to the problem under consideration for input I. For instance, when we study TSP then $\text{OPT}(P)$ will denote the length of a shortest tour on a point set P, and when we study the independent-set problem then $\text{OPT}(G)$ will denote the maximum size of any independent set of the input graph G. When no confusion can arise we will sometimes simply write OPT instead of $\text{OPT}(I)$.

A *minimization problem* is a problem where we want to find a solution with minimum value; TSP is an example of a minimization problem. An algorithm for a minimization problem is called a *ρ-approximation algorithm*, for some $\rho > 1$, if the algorithm produces for any input I a solution whose value is at most $\rho \cdot \text{OPT}(I)$. A *maximization problem* is a problem where we want to find a solution with maximum value; independent set is an example of a maximization problem. An algorithm for a maximization problem is called a *ρ-approximation*

[1]Chapter 36 of [CLRS] gives an introduction to the theory of NP-hardness.

algorithm, for some $\rho < 1$, if the algorithm produces for any input I a solution whose value is at least $\rho \cdot \text{OPT}(I)$. The factor ρ is called the *approximation factor* (or: *approximation ratio*) of the algorithm.[2]

The importance of lower bounds. It may seem strange that it is possible to prove that an algorithm is a ρ-approximation algorithm: how can we prove that an algorithm always produces a solution that is within a factor ρ of OPT when we do not know OPT? The crucial observation is that, even though we do not know OPT, we can often derive a *lower bound* (or, in the case of maximization problems: an upper bound) on OPT. If we can then show that our algorithm always produces a solution whose value is at most a factor ρ from the lower bound, then the algorithm is also within a factor ρ from OPT. Thus finding good lower bounds on OPT is an important step in the analysis of an approximation algorithm. In fact, the search for a good lower bound often leads to ideas on how to design a good approximation algorithm. This is something that we will see many times in the coming lectures.

6.1 Load balancing

Suppose we are given a collection of n jobs that must be executed. To execute the jobs we have m identical machines, $M_1, \ldots, M_m$, available. Executing job j on any of the machines takes time t_j, where $t_j > 0$. Our goal is to assign the jobs to the machines in such a way that the so-called *makespan*, the time until all jobs are finished, is as small as possible. Thus we want to spread the jobs over the machines as evenly as possible. Hence, we call this problem LOAD BALANCING.

Let's denote the collection of jobs assigned to machine M_i by $A(i)$. Then the *load* T_i of machine M_i—the total time for which M_i is busy—is given by

$$T_i = \sum_{j \in A(i)} t_j,$$

and the makespan of the assignment equals $\max_{1 \leqslant i \leqslant m} T_i$. The LOAD BALANCING problem is to find an assignment of jobs to machines that minimizes the makespan, where each job is assigned to a single machine. (We cannot, for instance, execute part of a job on one machine and the rest of the job on a different machine.) LOAD BALANCING is NP-hard.

Our first approximation algorithm for LOAD BALANCING is a straightforward greedy algorithm: we consider the jobs one by one and assign each job to the machine whose current load is smallest.

```
Algorithm Greedy-Scheduling($t_1, \ldots, t_n, m$)
1. Initialize $T_i \leftarrow 0$ and $A(i) \leftarrow \emptyset$ for $1 \leqslant i \leqslant m$.
2.     for $j \leftarrow 1$ to $n$
3.        do ▷ Assign job $j$ to the machine $M_k$ of minimum load
4.           Find a $k$ such that $T_k = \min_{1 \leqslant i \leqslant m} T_i$
5.           $A(k) \leftarrow A(k) \cup \{j\}$; $T_k \leftarrow T_k + t_j$
```

[2]In some texts the approximation factor ρ is required to be always greater than 1. For a maximization factor the solution should then have a value at least $(1/\rho) \cdot \text{OPT}$.

This algorithm clearly assigns each job to one of the m available machines. Moreover, it runs in polynomial time. In fact, if we maintain the loads T_i in a min-heap then we can find the machine k with minimum load in $O(1)$ time and update T_k in $O(\log m)$ time. This way the entire algorithm can be made to run in $O(n \log m)$ time. The main question is how good the assignment is. Does it give an assignment whose makespan is close to OPT? The answer is yes. To prove this we need a lower bound on OPT, and then we must argue that the makespan of the assignment produced by the algorithm is not much more than this lower bound.

There are two very simple observations that give a lower bound. First of all, the best one could hope for is that it is possible to spread the jobs perfectly over the machines so that each machine has the same load, namely $\sum_{1 \leqslant j \leqslant n} t_j/m$. In many cases this already provides a pretty good lower bound. When there is one very large job and all other jobs have processing time close to zero, however, then the upper bound is weak. In that case the trivial lower bound of $\max_{1 \leqslant j \leqslant n} t_j$ will be stronger. To summarize, we have

Lemma 6.1 $\textsc{opt} \geqslant \max\left(\frac{1}{m} \sum_{1 \leqslant j \leqslant n} t_j \ , \ \max_{1 \leqslant j \leqslant n} t_j \right)$.

Let's define $\textsc{lb} := \max\left(\frac{1}{m} \sum_{1 \leqslant j \leqslant n} t_j \ , \ \max_{1 \leqslant j \leqslant n} t_j \right)$ to be the lower bound provided by Lemma 6.1. With this lower bound in hand we can prove that our simple greedy algorithm gives a 2-approximation.

Theorem 6.2 *Algorithm Greedy-Scheduling is a 2-approximation algorithm.*

Proof. We must prove that *Greedy-Scheduling* always produces an assignment of jobs to machines such that the makespan T satisfies $T \leqslant 2 \cdot \textsc{opt}$. Consider an input $t_1, \ldots, t_n, m$. Let M_{i^*} be a machine determining the makespan of the assignment produced by the algorithm, that is, a machine such that at the end of the algorithm we have $T_{i^*} = \max_{1 \leqslant i \leqslant m} T_i$. Let j^* be the last job assigned to M_{i^*}. The crucial property of our greedy algorithm is that at the time job j^* is assigned to M_{i^*}, machine M_{i^*} is a machine with the smallest load among all the machines. So if T'_i denotes the load of machine M_i just before job j^* is assigned, then $T'_{i^*} \leqslant T'_i$ for all $1 \leqslant i \leqslant m$. It follows that

$$m \cdot T'_{i^*} \ \leqslant \ \sum_{1 \leqslant i \leqslant m} T'_i \ = \ \sum_{1 \leqslant j < j^*} t_j \ < \ \sum_{1 \leqslant j \leqslant n} t_j \ \leqslant \ m \cdot \textsc{lb}.$$

Hence, $T'_{i^*} < \textsc{lb}$ and we can derive

$$\begin{aligned} T_{i^*} &= t_{j^*} + T'_{i^*} \\ &\leqslant t_{j^*} + \textsc{lb} \\ &\leqslant \max\nolimits_{1 \leqslant j \leqslant n} t_j + \textsc{lb} \\ &\leqslant 2 \cdot \textsc{lb} \\ &\leqslant 2 \cdot \textsc{opt} \qquad \text{(by Lemma 6.1)} \end{aligned}$$

□

So this simple greedy algorithm is never more than a factor 2 from optimal. Can we do better? There are several strategies possible to arrive at a better approximation factor. One possibility could be to see if we can improve the analysis of *Greedy-Scheduling*. Perhaps we might be able to show that the approximation factor is in fact at most $c \cdot \textsc{lb}$ for some $c < 2$.

Another way to improve the analysis might be to use a stronger lower bound than the one provided by Lemma 6.1. (Note that if there are instances where $\text{LB} = \text{OPT}/2$ then an analysis based on this lower bound cannot yield a better approximation ratio than 2.)

It is, indeed, possible to prove a better approximation factor for the greedy algorithm described above: a more careful analysis shows that the approximation factor is in fact $(2 - \frac{1}{m})$, where m is the number of machines. This is tight for the given algorithm: for any m there are inputs such that *Greedy-Scheduling* produces an assignment of makespan $(2 - \frac{1}{m}) \cdot \text{OPT}$. Thus the approximation ratio is fairly close to 2, especially when m is large. So if we want to get an approximation ratio better than $(2 - \frac{1}{m})$, then we have to design a better algorithm.

A weak point of our greedy algorithm is the following. Suppose we first have a large number of small jobs and then finally a single very large job. Our algorithm will first spread the small jobs evenly over all machines and then add the large job to one of these machines. It would have been better, however, to give the large job its own machine and spread the small jobs over the remaining machines. Note that our algorithm would have produced this assignment if the large job would have been handled first. This observation suggest the following adaptation of the greedy algorithm: we first sort the jobs according to decreasing processing times, and then run *Greedy-Scheduling*. We call the new algorithm *Ordered-Scheduling*.

Does the new algorithm really have a better approximation ratio? The answer is yes. However, the lower bound provided by Lemma 6.1 is not sufficient to prove this; we also need the following lower bound.

Lemma 6.3 *Consider a set of n jobs with processing times $t_1, \ldots, t_n$ that have to be scheduled on m machines, where $t_1 \geqslant t_2 \geqslant \cdots \geqslant t_n$. If $n > m$, then $\text{OPT} \geqslant t_m + t_{m+1}$.*

Proof. Since there are m machines, at least two of the jobs $1, \ldots, m+1$, say jobs j and j', have to be scheduled on the same machine. Hence, the load of that machine is $t_j + t_{j'}$, which is at least $t_m + t_{m+1}$ since the jobs are sorted by processing times. □

Theorem 6.4 *Algorithm Ordered-Scheduling is a (3/2)-approximation algorithm.*

Proof. The proof is very similar to the proof of Theorem 6.2. Again we consider a machine M_{i^*} that has the maximum load, and we consider the last job j^* scheduled on M_{i^*}. If $j^* \leqslant m$, then j^* is the only job scheduled on M_{i^*}—this is true because the greedy algorithm schedules the first m jobs on different machines. Hence, our algorithm is optimal in this case. Now consider the case $j^* > m$. As in the proof of Theorem 6.2 we can derive

$$T_{i^*} \quad \leqslant \quad t_{j^*} + \tfrac{1}{m} \textstyle\sum_{1 \leqslant i \leqslant n} t_i.$$

The second term can be bounded as before using Lemma 6.1:

$$\tfrac{1}{m} \textstyle\sum_{1 \leqslant i \leqslant n} t_i \quad \leqslant \quad \max\left(\max_{1 \leqslant j \leqslant n} t_j \ , \ \tfrac{1}{m} \sum_{1 \leqslant i \leqslant n} t_i \right) \quad \leqslant \quad \text{OPT}.$$

For the first term we use that $j^* > m$. Since the jobs are ordered by processing time we have $t_{j^*} \leqslant t_{m+1} \leqslant t_m$. We can therefore use Lemma 6.3 to get

$$t_{j^*} \quad \leqslant \quad (t_m + t_{m+1})/2 \quad \leqslant \quad \text{OPT}/2.$$

Hence, the total load on M_{i^*} is at most $(3/2) \cdot \text{OPT}$. □

Chapter 7

The Traveling Salesman Problem

Let $G = (V, E)$ be an undirected graph. A *Hamiltonian cycle* of G is a cycle that visits every vertex $v \in V$ exactly once. Instead of Hamiltonian cycle, we sometimes also use the term *tour*. Not every graph has a Hamiltonian cycle: if the graph is a single path, for example, then obviously it does not have a Hamiltonian cycle. The problem HAMILTONIAN CYCLE is to decide for a given graph graph G whether it has a Hamiltonian cycle. HAMILTONIAN CYCLE is NP-complete.

Now suppose that G is a *complete graph*—that is, G has an edge between every pair of vertices—where each edge $e \in E$ has a non-negative length. It is easy to see that because G is complete it must have a Hamiltonian cycle. Since the edges now have lengths, however, some Hamiltonian cycles may be shorter than others. This leads to the *traveling salesman problem*, or TSP for short: given a complete graph $G = (V, E)$, where each edge $e \in E$ has a length, find a minimum-length tour (Hamiltonian cycle) of G. (The length of a tour is defined as the sum of the lengths of the edges in the tour.) TSP is NP-hard. We are therefore interested in approximation algorithms. Unfortunately, even this is too much to ask.

Theorem 7.1 *There is no value c for which there exists a polynomial-time c-approximation algorithm for* TSP*, unless P=NP.*

Proof. As noted above, HAMILTONIAN CYCLE is NP-complete, so there is no polynomial-time algorithm for the problem unless P=NP. Let c be any value. We will prove that if there is a polynomial time c-approximation algorithm for TSP, then we can also solve HAMILTONIAN CYCLE in polynomial time. The theorem then follows.

Let $G = (V, E)$ be a graph for which we want to decide if it admits a Hamiltonian cycle. We construct a complete graph $G^* = (V, E^*)$ from G as follows. For every pair of vertices u, v we put an edge in E^*, where we set $\textit{length}((u, v)) = 1$ if $(u, v) \in E$ and $\textit{length}((u, v)) = c \cdot |V| + 1$ if $(u, v) \notin E$. The graph G^* can be constructed from G in polynomial time—in $O(|V|^2)$ time, to be precise. Let $\text{OPT}(G^*)$ denote the minimum length of any tour for G^*.

Now suppose we have a c-approximation algorithm $\mathcal{A}$ for TSP. Run $\mathcal{A}$ on G^*. We claim that $\mathcal{A}$ returns a tour of length $|V|$ if and only if G has a Hamiltonian cycle. For the "if"-part we note that $\text{OPT}(G^*) = |V|$ if G has a Hamiltonian cycle, since then there is a tour in G^* that only uses edges of length 1. Since $\mathcal{A}$ is a c-approximation algorithm it must return a tour of length at most $c \cdot \text{OPT}(G^*) = c \cdot |V|$. Obviously such a tour cannot use any edges of length $c \cdot |V| + 1$ and so it only uses edges that were already in G. In other words, if G has a Hamiltonian cycle then $\mathcal{A}$ returns tour of length $|V|$. For the "only if"-part, suppose

$\mathcal{A}$ returns a tour of length $|V|$. Then obviously that tour can only use edge of length 1—in other words, edges from E—which means G has a Hamiltonian cycle. $\square$

Note that in the proof we could also have set the lengths of the edges in E to 0 and the lengths of the other edges to 1. Then $\text{OPT}(G^*) = 0$ if and only if G has a Hamiltonian cycle. When $\text{OPT}(G^*) = 0$, then $c \cdot \text{OPT}(G^*) = 0$ no matter how large c is. Hence, any approximation algorithm must solve the problem exactly. In some sense, this is cheating: when $\text{OPT} = 0$ allowing a (multiplicative) approximation factor does not help, so it is not surprising that one cannot get a polynomial-time approximation algorithm (unless P=NP). The proof above shows that this is even true when all edge lengths are positive, which is a stronger result.

This is disappointing news. But fortunately things are not as bas as they seem: when the edge lengths satisfy the so-called *triangle inequality* then we *can* obtain good approximation algorithms. The triangle inequality states that for every three vertices u, v, w we have

$$\mathit{length}((u,w)) \leqslant \mathit{length}((u,v)) + \mathit{length}((v,w)).$$

In other words, it is not more expensive to go directly from u to w than it is to go via some intermediate vertex v. This is a very natural property. It holds for instance for *Euclidean TSP*. Here the vertices in V are points in the plane (or in some higher-dimensional space), and the length of an edge between two points is the Euclidean distance between them. As we will see below, for graphs whose edge lengths satisfy the triangle inequality, it is fairly easy to give a 2-approximation algorithm. With a little more effort, we can improve the approximation factor to 3/2. For the special case of Euclidean TSP there is even a PTAS; this algorithm is fairly complicated, however, and we will not discuss it here. We will use the following property of graphs whose edge lengths satisfy the triangle inequality.

Observation 7.2 *Let $G = (V, E)$ be a graph whose edge lengths satisfy the triangle inequality, and let $v_1, v_2, \ldots, v_k$ be any path in G. Then $\mathit{length}((v_1, v_k)) \leqslant \mathit{length}(v_1, v_2, \ldots, v_k)$.*

Proof. By induction on k. If $k = 2$ the statement is trivially true, so assume $k > 2$. By the induction hypothesis, we know that $\mathit{length}((v_1, v_{k-1})) \leqslant \mathit{length}(v_1, v_2, \ldots, v_{k-1})$. Moreover, $\mathit{length}((v_1, v_k)) \leqslant \mathit{length}((v_1, v_{k-1})) + \mathit{length}((v_{k-1}, v_k))$ by the triangle inequality. Hence,

$$\begin{aligned}
\mathit{length}((v_1, v_k)) &\leqslant \mathit{length}((v_1, v_{k-1})) + \mathit{length}((v_{k-1}, v_k) \\
&\leqslant \mathit{length}(v_1, v_2, \ldots, v_{k-1}) + \mathit{length}((v_{k-1}, v_k) \\
&= \mathit{length}(v_1, v_2, \ldots, v_k).
\end{aligned}$$

$\square$

7.1 A simple 2-approximation algorithm

A *spanning tree* of a graph G is a tree—a connected acyclic graph—whose vertex set is V; a *minimum spanning tree* of a graph (whose edges have lengths) is a spanning tree whose total edge length is minimum among all spanning trees for G. Spanning trees and tours seem very similar: both are subgraphs of G that connect all vertices. The only difference is that in a spanning tree the connections form a tree, while in a tour they form a cycle. From a computational point of view, however, this makes a huge difference: while TSP is NP-hard,

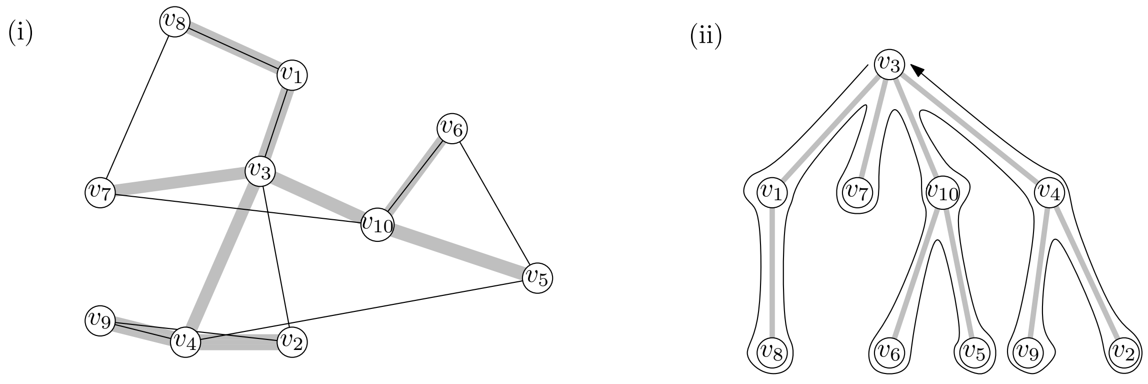

Figure 7.1: (i) A spanning tree (thick grey) and the tour (thin black) that is found when the traversal shown in (ii) is used. (ii) Possible inorder traversal of the spanning tree in (i). The traversal shown results from choosing v_3 as the root vertex and visiting the children in some specific order. Note that a different tour would result if we visit the children in a different order.

computing a minimum spanning tree can be done in polynomial time with a simple greedy algorithm such as Kruskal's algorithm or Prim's algorithm—see [CLRS] for details.

Now let $G = (V, E)$ be a complete graph whose edge lengths satisfy the triangle inequality. As usual, we will derive our approximation algorithm for TSP from an efficiently computable lower bound. In this case the lower bound is provided by the minimum spanning tree.

Lemma 7.3 *Let* OPT *denote the minimum length of any tour of the given graph G, and let* MST *denote the total length of a minimum spanning tree of G. Then* OPT $\geqslant$ MST.

Proof. Let Γ be an optimal tour for G. By deleting an edge from Γ we obtain a path Γ' and since all edge lengths are non-negative we have $\mathit{length}(\Gamma') \leqslant \mathit{length}(\Gamma) =$ OPT. Because a path is (a special case of) a tree, a minimum spanning tree is at least as short as Γ'. Hence, MST $\leqslant \mathit{length}(\Gamma') \leqslant$ OPT. $\square$

So the length of a minimum spanning tree provides a lower bound on the minimum length of a tour. But can we also use a minimum spanning tree to compute an approximation of a minimum-length tour? The answer is yes: we simply make an arbitrary vertex of the minimum spanning tree $\mathcal{T}$ to be the root and use an inorder traversal of $\mathcal{T}$ to get the tour. (An inorder traversal of a rooted tree is a traversal that starts at the root and then proceeds as follows. Whenever a vertex is reached, we first visit that vertex and then we recursively visit each of its subtrees.) Figure 7.1 illustrates this. Thus our approximation algorithm is as follows.

```
Algorithm ApproxTSP($G$)
1.  Compute a minimum spanning tree $\mathcal{T}$ for $G$.
2.          Turn $\mathcal{T}$ into a rooted tree by selecting an arbitrary vertex $u \in \mathcal{T}$ as the root.
3.          Initialize an empty tour $\Gamma$.
4.          InorderTraversal($u, \Gamma$)
5.          Append $u$ to $\Gamma$, so that the tour returns to the starting vertex.
6.          return $\Gamma$
```

Below is the algorithm for the inorder traversal of a the minimum-spanning tree $\mathcal{T}$. Recall that we have assigned an arbitrary vertex u as the root of $\mathcal{T}$, where the traversal is started. This also determines for each edge (u, v) of $\mathcal{T}$ whether v is a child a u or vice versa.

```
Algorithm InorderTraversal($u, \Gamma$)
1. Append $u$ to $\Gamma$.
2.     for each child $v$ of $u$
3.         do InorderTraversal($v$)
```

Theorem 7.4 *ApproxTSP is a 2-approximation algorithm.*

Proof. Let Γ denote the tour reported by the algorithm, and let MST denote the total length of the minimum spanning tree. We will prove that $\mathit{length}(\Gamma) \leqslant 2 \cdot \text{MST}$. The theorem then follows from Lemma 7.3.

Consider an inorder traversal of the minimum spanning tree $\mathcal{T}$ where we change line 3 to

```
3.     do InorderTraversal($v$); Append $u$ to $\Gamma$.
```

In other words, after coming back from recursively visiting a subtree of a node u we first visit u again before we move on to the next subtree of u. This way we get a cycle Γ' where some vertices are visited more than once, and where every edge in Γ' is also an edge in $\mathcal{T}$. In fact, every edge in $\mathcal{T}$ occurs exactly twice in Γ', so $\mathit{length}(\Gamma') = 2 \cdot \text{MST}$. The tour Γ can be obtained from Γ' by deleting vertices so that only the first occurrence of each vertex remains. This means that certain paths $v_1, v_2, \ldots, v_k$ are shortcut by the single edge (v_1, v_k). By Observation 7.2, all the shortcuts are at most as long as the paths they replace, so $\mathit{length}(\Gamma) \leqslant \mathit{length}(\Gamma') \leqslant 2 \cdot \text{MST}$. □

Is this analysis tight? Unfortunately, the answer is basically yes: there are graphs for which the algorithm produces a tour of length $(2 - \frac{1}{|V|}) \cdot \text{OPT}$, so when $|V|$ gets larger and larger the worst-case approximation ratio gets arbitrarily close to 2. Hence, if we want to improve the approximation ratio, we have to come up with a different algorithm. This is what we do in the next section.

7.2 Christofides's (3/2)-approximation algorithm

Our improved approximation algorithm, which is due to Nicos Christofides, also starts by computing a minimum spanning tree. The difference with our 2-approximation algorithm is that the shortcuts are chosen more cleverly. This is done based on the following two facts.

An *Euler tour* of an undirected graph is a cycle that visits every edge exactly once; note that it may visit a vertex more than once.[1] Determining whether a graph has a Hamiltonian cycle is hard, but determining whether it has an Euler tour is quite easy: a connected undirected graph has an Euler tour if and only if the degree of every vertex is even. Moreover, it is not only easy to determine if a graph has an Euler tour, it is also easy to compute one if it exists.

Of course a minimum spanning tree—or any other tree for that matter—does not admit an Euler tour, since the leaves of the tree have degree 1. The idea is therefore to add extra

[1]This terminology is standard but perhaps a bit unfortunate, since an Euler tour is not necessarily a tour under the definition we gave earlier.

edges to the minimum spanning tree such that all vertices have even degree, and then take an Euler tour of the resulting graph. To this end we need the concept of so-called matchings.

Let G be a graph with an even number of vertices. Then a *matching* is a collection M of edges from the graph such that every vertex is the endpoint of at most one edge in M. The matching is called *perfect* if every vertex is incident to exactly one edge in M, and if its total edge length is minimum among all perfect matchings then we call M a *minimum perfect matching*. It is known that a minimum perfect matching of a complete graph with an even number of vertices can be computed in polynomial time. (Notice that a complete graph with an even number of vertices always has a perfect matching.)

Lemma 7.5 *Let $G = (V, E)$ be a graph and let $V^* \subset V$ be any subset of an even number of vertices. Let* $\textsc{opt}$ *denote the minimum length of any tour on G, and let M^* be a perfect matching on the complete graph $G^* = (V^*, E^*)$, where the lengths of the edges in E^* are equal to the lengths of the corresponding edges in E. Then $\mathit{length}(M^*) \leqslant \frac{1}{2} \cdot \textsc{opt}$.*

Proof. Let $\Gamma = v_1, \ldots, v_n, v_1$ be an optimal tour. Let's first assume that $V^* = V$. Consider the following two perfect matchings: $M_1 = \{(v_1, v_2), (v_3, v_4), \ldots, (v_{n-1}, v_n)\}$ and $M_2 = \{(v_2, v_3), (v_4, v_5), \ldots, (v_n, v_1)\}$. Then $\mathit{length}(M_1) + \mathit{length}(M_2) = \mathit{length}(\Gamma) = \textsc{opt}$. Hence, for the minimum-length perfect matching M^* we have

$$\mathit{length}(M^*) \leqslant \min(\mathit{length}(M_1), \mathit{length}(M_2)) \leqslant \textsc{opt}/2.$$

If $V^* \neq V$ then we can use basically the same argument: Let $n^* = |V^*|$ and number the vertices from V^* as $v_1^*, \ldots, v_{n^*}^*$ in the order they are encountered by Γ. Consider the two matchings $M_1 = \{(v_1^*, v_2^*), \ldots, (v_{n^*-1}, v_{n^*})\}$ and $M_2 = \{(v_2^*, v_3^*), \ldots, (v_{n^*}, v_1)\}$. One of these has length at most $\mathit{length}(\Gamma^*)/2$, where Γ^* is the tour $v_1^*, \ldots, v_{n^*}^*, v_1^*$. The result follows because $\mathit{length}(\Gamma^*) \leqslant \mathit{length}(\Gamma)$ by the triangle inequality. □

The algorithm is now as follows.

```
Algorithm ChristofidesTSP($G$)
1. Compute a minimum spanning tree $\mathcal{T}$ for $G$.
2.     Let $V^* \subset V$ be the set of vertices of odd degree in $\mathcal{T}$.
3.     Compute a minimum perfect matching $M$ on the complete graph $G^* = (V^*, E^*)$.
4.     Add the edges from $M$ to $\mathcal{T}$, and find an Euler tour $\Gamma$ of the resulting (multi-
       )graph.
5.     For each vertex that occurs more than once in $\Gamma$, remove all but one of its occur-
       rences.
6.     return $\Gamma$
```

Theorem 7.6 *ChristofidesTSP is a (3/2)-approximation algorithm.*

Proof. First we note that in any graph, the number of odd-degree vertices must be even—this is easy to show by induction on the number of edges. Hence, the set V^* has an even number of vertices, so it admits a perfect matching. Adding the edges from the matching M to the tree $\mathcal{T}$ ensures that every vertex of odd degree gets an extra incident edge, so all degrees become even. (Note that the matching M may contain edges that were already present in $\mathcal{T}$. Hence, after we add these edges to M we in fact have a *multi-graph*. But this is not a problem for the rest of the algorithm.) It follows that after adding the edges from M, we get a multi-graph that has an Euler tour Γ. The length of Γ is at most $\mathit{length}(\mathcal{T}) + \mathit{length}(M)$, which is at most $(3/2) \cdot \textsc{opt}$ by Lemmas 7.3 and 7.5. By Observation 7.2, removing the superfluous occurrences of the vertices occurring more than once in line 5 can only decrease the length of the tour. $\square$

Christofides's algorithm is still the best known algorithm for TSP for graphs satisfying the triangle inequality. For Euclidean TSP, however, S. Aurora developed a PTAS. As mentioned earlier, this PTAS is rather complicated and we will not discuss it here.

Chapter 8

Approximation via LP Rounding

Let $G = (V, E)$ be an (undirected) graph. A subset $C \subset V$ is called a *vertex cover* for G if for every edge $(v_i, v_j) \in E$ we have $v_i \in C$ or $v_j \in C$ (or both). In other words, for every edge in E at least one of its endpoints is in C.

8.1 Unweighted vertex cover

The unweighted version of the Vertex Cover problem is to find a vertex cover of minimum size for a given graph G. This problem is NP-complete.

Let's try to come up with an approximation algorithm. A natural greedy approach would be the following. Initialize the cover C as the empty set, and set $E' := E$; the set E' will contain the edges that are not yet covered by C. Now take an edge $(v_i, v_j) \in E'$, put one of its two vertices, say v_i, into C, and remove from E' all edges incident to v_i. Repeat the process until $E' = \emptyset$. Clearly C is a vertex cover after the algorithm has finished. Unfortunately the algorithm has a very bad approximation ratio: there are instances where it can produce a vertex cover of size $|V| - 1$ even though a vertex cover of size 1 exists.

A small change in the algorithm leads to a 2-approximation algorithm. The change is based on the following lower bound. Call two edges $e, e' \in E$ *adjacent* if they share an endpoint.

Lemma 8.1 *Let $G = (V, E)$ be a graph and let* OPT *denote the minimum size of a vertex cover for G. Let $E^* \subset E$ be any subset of edges such that no two edges in E^* are adjacent. Then* OPT $\geqslant |E^*|$.

Proof. Let C be an optimal vertex cover for G. By definition, any edge $e \in E^*$ must be covered by a vertex in C, and since the edges in E^* are non-adjacent any vertex in C can cover at most one edge in E^*. □

This lemma suggests the following greedy algorithm.

```
Algorithm ApproxVertexCover(V, E)
1.  $C \leftarrow \emptyset$; $E' \leftarrow E$
2.          ▷ Invariant: $C$ is a vertex cover for the graph $G' = (V, E \setminus E')$
3.          while $E' \neq \emptyset$
4.              do Take an arbitrary edge $(v_i, v_j) \in E'$.
5.                  $C \leftarrow C \cup \{v_i, v_j\}$
```

```
6.              Remove all edges from $E'$ that are adjacent to $(v_i, v_j)$.
7.      return $C$
```

It is easy to check that the **while**-loop indeed maintains the invariant. After the **while**-loop has terminated—the loop must terminate since at every step we remove at least one edge from E'—we have $E \setminus E' = E \setminus \emptyset = E$. Together with the invariant this implies that the algorithm indeed returns a vertex cover. Next we show that the algorithm gives a 2-approximation.

Theorem 8.2 *Algorithm ApproxVertexCover produces a vertex cover C such that $|C| \leqslant 2 \cdot \text{OPT}$, where* OPT *is the minimum size of a vertex cover.*

Proof. Let E^* be the set of edges selected in line 4 over the course of the algorithm. Then C consists of the endpoints of the edges in E^*, and so $|C| \leqslant 2|E^*|$. Moreover, no two edges in E^* are adjacent because as soon as an edge (v_i, v_j) is selected from E' all edges in E' adjacent to (v_i, v_j) are removed from E'. The theorem now follows from Lemma 8.1. □

8.2 Weighted Vertex Cover

Now let's consider a generalization of Vertex Cover, where each vertex $v_i \in V$ has a weight w_i and we want to find a vertex cover of minimum total weight. We call the new problem Weighted Vertex Cover. The first idea that comes to mind to get an approximation algorithm for Weighted Vertex Cover is to generalize *ApproxVertexCover* as follows: instead of selecting an arbitrary edge (v_i, v_j) from E' in line 4, we select the edge of minimum weight (where the weight of an edge is defined as the sum of the weights of its endpoints). Unfortunately this doesn't work: the weight of the resulting cover can be arbitrarily much larger than the weight of an optimal cover. Our new approach will be based on linear programming.

(Integer) linear programming. In a linear-programming problem we are given a linear *cost function* of d real variables $x_1, \ldots, x_d$ and a set of n linear *constraints* on these variables. The goal is to assign values to the variables so that the cost function is minimized (or: maximized) and all constraints are satisfied. In other words, the Linear Programming problem can be stated as follows:

$$\begin{array}{ll} \text{Minimize} & c_1x_1 + \cdots + c_dx_d \\ \text{Subject to} & a_{1,1}x_1 + \cdots + a_{1,d}x_d \leqslant b_1 \\ & a_{2,1}x_1 + \cdots + a_{2,d}x_d \leqslant b_2 \\ & \vdots \\ & a_{n,1}x_1 + \cdots + a_{n,d}x_d \leqslant b_n \end{array}$$

There are algorithms—for example the so-called *interior-point methods*—that can solve linear programs in time polynomial in the input size.[1] In practice linear programming is often done with with famous *simplex method*, which is exponential in the worst case but works quite well

[1]Here the input size is measured in terms of the number of bits needed to describe the input, so this is different from the usual notion of input size.

in most practical applications. Hence, if we can formulate a problem as a linear-programming problem then we can solve it efficiently, both in theory and in practice.

There are several problems that can be formulated as a linear-programming problem but with one twist: the variables $x_1, \ldots, x_d$ can not take on real values but only integer values. This is called INTEGER LINEAR PROGRAMMING. (When the variables can only take the values 0 or 1, the problem is called 0/1 LINEAR PROGRAMMING.) Unfortunately, INTEGER LINEAR PROGRAMMING and 0/1 LINEAR PROGRAMMING are considerably harder than LINEAR PROGRAMMING. In fact, INTEGER LINEAR PROGRAMMING and 0/1 LINEAR PROGRAMMING are NP-complete. However, formulating a problem as an integer linear program can still be useful, as shall see next.

Approximating via LP relaxation and rounding. Let's go back to WEIGHTED VERTEX COVER. Thus we are given a weighted graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$, and we want to find a minimum-weight vertex cover. To formulate this as a 0/1 linear program, we introduce a variable x_i for each vertex $v_i \in V$; the idea is to have $x_i = 1$ if v_i is taken into the vertex cover, and $x_1 = 0$ if v_i is not taken into the cover. When is a subset $C \subset V$ is vertex cover? Then C must contain at least one endpoint for every edge (v_i, v_j). This means we must have $x_i = 1$ or $x_j = 1$. We can enforce this by introducing for every edge $(v_i, v_j) \in E$ the constraint $x_i + x_j \geqslant 1$. Finally, we wish to minimize the total weight of the cover, so we get as a cost function $\sum_{i=1}^{n} w_i x_i$. To summarize, solving the weighted vertex-cover problem corresponds to solving the following 0/1 linear-programming problem.

$$\begin{aligned} \text{Minimize} \quad & w_1 x_1 + \cdots + w_n x_n \\ \text{Subject to} \quad & x_i + x_j \geqslant 1 \qquad \text{for all edges } (v_i, v_j) \in E \\ & x_i \in \{0, 1\} \qquad \text{for } 1 \leqslant i \leqslant n \end{aligned} \tag{8.1}$$

As noted earlier, solving 0/1 linear programs is hard. Therefore we perform *relaxation*: we drop the restriction that the variables can only take integer values and we replace the integrality constraints (8.1) by

$$0 \leqslant x_i \leqslant 1 \qquad \text{for } 1 \leqslant i \leqslant n \tag{8.2}$$

This linear program can be solved in polynomial time. But what good is a solution where the variables can take on any real number in the interval $[0, 1]$? A solution with $x_i = 1/3$, for instance, would suggest that we put 1/3 of the vertex v_i into the cover—something that does not make sense. First we note that the solution to our new relaxed linear program provides us with a lower bound.

Lemma 8.3 *Let W denote the value of an optimal solution to the relaxed linear program described above, and let* OPT *denote the minimum weight of a vertex cover. Then* OPT $\geqslant W$.

Proof. Any vertex cover corresponds to a feasible solution of the linear program, by setting the variables of the vertices in the cover to 1 and the other variables to 0. Hence, the optimal solution of the linear program is at least as good as this solution. (Stated differently: we already argued that an optimal solution of the 0/1-version of the linear program corresponds to an optimal solution of the vertex-cover problem. Relaxing the integrality constraints clearly cannot make the solution worse.) □

The next step is to derive a valid vertex cover—or, equivalently, a feasible solution to the 0/1 linear program—from the optimal solution to the relaxed linear program. We want to do this in such a way that the total weight of the solution does not increase by much. This can simply be done by *rounding*: variables whose value is at least 1/2 are rounded to 1, variables whose value is less than 1/2 are rounded to 0. We thus obtain the following algorithm for Weighted Vertex Cover.

```
Algorithm ApproxWeightedVertexCover(V, E)
1.  ▷ $G = (V, E)$ is a graph where each vertex $v_i \in V$ $(1 \leqslant i \leqslant n)$ has weight $w_i$.
2.      Solve the relaxed linear program corresponding to the given problem:
            Minimize    $w_1x_1 + \cdots + w_nx_n$
            Subject to  $x_i + x_j \geqslant 1$        for all edges $(v_i, v_j) \in E$
                        $0 \leqslant x_i \leqslant 1$        for $1 \leqslant i \leqslant n$
3.      $C \leftarrow \{v_i \in V : x_i \geqslant 1/2\}$
4.      return $C$
```

Theorem 8.4 *Algorithm ApproxWeightedVertexCover is a 2-approximation algorithm.*

Proof. We first argue that the set C returned by the algorithm is a vertex cover. Consider an edge $(v_i, v_j) \in E$. Then $x_i + x_j \geqslant 1$ is one of the constraints of the linear program. Hence, the reported solution to the linear program—note that a solution will be reported, since the program is obviously feasible by setting all variables to 1—has $\max(x_i, x_j) \geqslant 1/2$. It follows that at least one of v_i and v_j will be put into C.

Let $W := \sum_{i=1}^{n} w_i x_i$ be the total weight of the optimal solution to the relaxed linear program. By Lemma 8.5 we have $\text{OPT} \geqslant W$. Using that $x_i \geqslant 1/2$ for all $v_i \in C$, we can now bound the total weight of C as follows:

$$\sum_{v_i \in C} w_i \;\leqslant\; \sum_{v_i \in C} w_i \cdot 2x_i \;\leqslant\; 2\sum_{v_i \in C} w_i x_i \;\leqslant\; 2\sum_{i=1}^{n} w_i x_i \;=\; 2W \;\leqslant\; 2 \cdot \text{OPT}$$

□

Note that, as always, the approximation ratio of our algorithm is obtained by comparing the obtained solution to a certain lower bound—in this case the solution to the LP relaxation. The worst-case ratio between the solution to the integer linear program (which models the problem exactly) and its relaxed version is called the *integrality gap*. For approximation algorithms based on rounding the relaxation of an integer linear program, one typically cannot prove a better approximation ratio than the integrality gap.

8.3 Set Cover

Let $Z := \{z_1, \ldots, z_m\}$ be a finite set. A *set cover* for Z is a collection of subsets of Z whose union is Z. The SET COVER problem is, given a set Z and a collection $\mathcal{S} = S_1, \ldots, S_n$ of subsets of Z, to select a minimum number of subsets from $\mathcal{S}$ that together form a set cover for Z.

SET COVER is a generalization of VERTEX COVER. This can be seen as follows. Let $G = (V, E)$ be the graph for which we want to obtain a vertex cover. We can construct an instance of SET COVER from G as follows: The set Z is the set of edges of G, and every vertex $v_i \in V$ defines a subset S_i consisting of those edges of which v_i is an endpoint. Then a set cover for the input $Z, S_1, \ldots, S_n$ corresponds to a vertex cover for G. (Note that SET COVER is more general than VERTEX COVER, because in the instance of SET COVER defined by a VERTEX COVER instance, every element occurs in exactly two sets—in the general problem an element from Z can occur in many subsets.) In WEIGHTED SET COVER every subset S_i has a weight w_i and we want to find a set cover of minimum total weight.

In this section we will develop an approximation algorithm for Weighted Set Cover. To this end we first formulate the problem as a 0/1 linear program: we introduce a variable x_i that indicates whether S_i is in the cover ($x_i = 1$) or not ($x_i = 0$), and we introduce a constraint for each element $z_j \in Z$ that guarantees that z_j will be in at least one of the chosen sets. The constraint for z_j is defined as follows. Let

$$\mathcal{S}(j) := \{i : 1 \leqslant i \leqslant n \text{ and } z_j \in S_i\}.$$

Then one of the chosen sets contains z_j if and only if $\sum_{i \in \mathcal{S}(j)} x_i \geqslant 1$. This leads to the following 0/1 linear program.

$$\begin{array}{lll} \text{Minimize} & w_1 x_1 + \cdots + w_n x_n & \\ \text{Subject to} & \sum_{i \in \mathcal{S}(j)} x_i \geqslant 1 & \text{for all } 1 \leqslant j \leqslant m \\ & x_i \in \{0,1\} & \text{for } 1 \leqslant i \leqslant n \end{array} \tag{8.3}$$

We relax this 0/1 linear program by replacing the integrality constraints in (8.3) by the following constraints:

$$0 \leqslant x_i \leqslant 1 \qquad \text{for } 1 \leqslant i \leqslant n \tag{8.4}$$

We obtain a linear program that we can solve in polynomial time. As in the case of Weighted Vertex Cover, the value of an optimal solution to this linear program is a lower bound on the value of an optimal solution to the 0/1 linear program and, hence, a lower bound on the minimum total weight of a set cover for the given instance:

Lemma 8.5 *Let W denote the value of an optimal solution to the relaxed linear program described above, and let* OPT *denote the minimum weight of a set cover. Then* OPT $\geqslant W$.

The next step is to use the solution to the linear program to obtain a solution to the 0/1 linear program (or, in other words, to the set cover problem). Rounding in the same way as for the vertex cover problem—rounding variables that are at least 1/2 to 1, and the other variables to 0—does not work: such a rounding scheme will not give a set cover. Instead we use the following *randomized rounding* strategy:

For each S_i independently, put S_i into the cover C with probability x_i.

Lemma 8.6 *The expected total weight of C is at most* OPT.

Proof. By definition, the total weight of C is the sum of the weights of its subsets. Let's define an indicator random variable Y_i that tells us whether a set S_i is in the cover C:

$$Y_i = \begin{cases} 1 & \text{if } S_i \in C \\ 0 & \text{otherwise} \end{cases}$$

We have

$$\begin{array}{rcll} \mathrm{E}[\text{ weight of } C\ ] & = & \mathrm{E}[\ \sum_{i=1}^{n} w_i Y_i\ ] & \\ & = & \sum_{i=1}^{n} w_i\, \mathrm{E}[Y_i] & \text{(by linearity of expectation)} \\ & = & \sum_{i=1}^{n} w_i \cdot \Pr[\ S_i \text{ is put into } C\ ] & \\ & = & \sum_{i=1}^{n} w_i x_i & \\ & \leqslant & \text{OPT} & \text{(by Lemma 8.5)} \end{array}$$

□

So the total weight of C is very good. Is C is valid set cover? To answer this question, let's look at the probability that an element $z_j \in Z$ is not covered. Recall that $\sum_{i \in \mathcal{S}(j)} x_i \geqslant 1$. Suppose that z_j is present in ℓ subsets, that is, $|\mathcal{S}(j)| = \ell$. To simplify the notation, let's renumber the sets such that $\mathcal{S}(j) = \{1, \ldots, \ell\}$. Then we have

$$\Pr[\ z_j \text{ is not covered }\ ] \ = \ (1 - x_1) \cdot \cdots \cdot (1 - x_\ell) \ \leqslant \ (1 - \frac{1}{\ell})^\ell,$$

where the last inequality follows from the fact that $(1 - x_1) \cdot \cdots \cdot (1 - x_\ell)$ is maximized when the x_i's sum up to exactly 1 and are evenly distributed, that is, when $x_i = 1/\ell$ for all i. Since $(1 - (1/\ell))^\ell \leqslant 1/e$, where $e \approx 2.718$ is the base of the natural logarithm, we conclude that

$$\Pr[\ z_j \text{ is not covered }\ ] \ \leqslant \ \frac{1}{e} \ \approx \ 0.268.$$

So the probability that any element z_j is covered is fairly high. But this is not good enough: there are many elements z_j and even though each one of them has a good chance of being covered, we cannot expect all of them to be covered simultaneously. (This is only to be expected, of course, since WEIGHTED SET COVER is NP-complete, so we shouldn't hope to find an optimal solution in polynomial time.) What we need is that each element z_j is covered *with high probability*. To this end we simply repeat the above procedure t times, for a suitable value of t: we generate covers $C_1, \ldots, C_t$ where each C_s is obtained using the randomized rounding strategy, and we take $C^* := C_1 \cup \cdots \cup C_t$ as our cover. Our final algorithm is thus as follows.

Algorithm *ApproxWeightedSetCover*$(X, \mathcal{S})$

```
1. ▷ $X = \{z_1, \ldots, z_m\}$, and $\mathcal{S} = \{S_1, \ldots, S_n\}$, and set $S_i \in \mathcal{S}$ $(1 \leqslant i \leqslant n)$ has weight $w_i$.
2.      Solve the relaxed linear program corresponding to the given problem:

            Minimize     $w_1 x_1 + \cdots + w_n x_n$
            Subject to   $\sum_{i \in \mathcal{S}(j)} x_i \geqslant 1$     for all $1 \leqslant j \leqslant m$
                         $0 \leqslant x_i \leqslant 1$                      for $1 \leqslant i \leqslant n$

3.      $t \leftarrow 2 \ln m$
4.      for $s \leftarrow 1$ to $t$
5.          do ▷ Compute $C_s$ by randomized rounding
6.              for $i \leftarrow 1$ to $n$
7.                  do Put $S_i$ into $C_s$ with probability $x_i$
8.      $C^* \leftarrow C_1 \cup \cdots \cup C_t$
9.      return $C^*$
```

Theorem 8.7 *Algorithm ApproxWeightedSetCover computes a collection C^* that is a set cover with probability at least $1 - 1/m$ and whose expected total weight is $O(\text{OPT} \cdot \log m)$.*

Proof. The expected weight of each C_s is at most OPT, so the expected total weight of C^* is at most $t \cdot \text{OPT} = O(\text{OPT} \cdot \log m)$. What is the probability that some fixed element z_j is not covered by any of the covers C_s? Since the covers C_s are generated independently, and each C_s fails to cover z_j with probability at most $1/e$, we have

$$\Pr[\ z_j \text{ is not covered by any } C_s\ ] \leqslant (1/e)^t.$$

Since $t = 2\ln m$ we conclude that z_j is not covered with probability at most $1/m^2$. Hence,

$$\begin{aligned}
\Pr[\text{ all elements } z_j \text{ are covered by } C^* \,] &= 1 - \Pr[\text{ at least one element } z_j \text{ is not covered by } C^* \,] \\
&\leqslant 1 - \textstyle\sum_{j=1}^{m} \Pr[\; z_j \text{ is not covered by } C^* \,] \\
&\leqslant 1 - 1/m
\end{aligned}$$

□

Chapter 9

Polynomial-time approximation schemes

When faced with an NP-hard problem one cannot expect to find a polynomial-time algorithm that always gives an optimal solution. Hence, one has to settle for an approximate solution. Of course one would prefer that the approximate solution is very close optimal, for example at most 5% worse. In other words, one would like to have an approximation ratio very close to 1. The approximation algorithms we have seen so far do not quite achieve this: for Load Balancing we gave an algorithm with approximation ratio 3/2, for Weighted Vertex Cover we gave an algorithm with approximation ratio 2, and for Weighted Set Cover the approximation ratio was even $O(\log n)$. Unfortunately it is not always possible to get a better approximation ratio: for some problems one can prove that it is not only NP-hard to solve the problem exactly, but that there is a constant $c > 1$ such that there is no polynomial-time c-approximation algorithm unless P=NP. Vertex Cover, for instance, cannot be approximated to within a factor 1.3606... unless P=NP, and for Set Cover one cannot obtain a better approximation factor than $\Theta(\log n)$.

Fortunately there are also problems where much better solutions are possible. In particular, some problems admit a so-called *polynomial-time approximation scheme*, or *PTAS* for short. Such an algorithm works as follows. Its input is, of course, an instance of the problem at hand, but in addition there is an input parameter $\varepsilon > 0$. The output of the algorithm is then a solution whose value is at most $(1+\varepsilon)\cdot \textsc{opt}$ (for a minimization problem) or at least $(1-\varepsilon)\cdot \textsc{opt}$ (for a maximization problem). The running time of the algorithm should be polynomial in n; its dependency on ε can be exponential however. So the running time can be $O(2^{1/\varepsilon}n^2)$ for example, or $O(n^{1/\varepsilon})$, or $O(n^2/\varepsilon)$, etc. If the dependency on the parameter $1/\varepsilon$ is also polynomial then we speak of a *fully polynomial-time approximation scheme (FPTAS)*. In this lecture we give an example of an FPTAS.

9.1 Knapsack

The Knapsack problem is defined as follows. We are given a set $X = \{x_1, \ldots, x_n\}$ of n items that each have a (positive) *weight* and a (positive) *profit*. The weight and profit of x_i are denoted by $\textit{weight}(x_i)$ and $\textit{profit}(x_i)$, respectively. Moreover, we have a knapsack that can carry items of total weight W. For a subset $S \subset X$, define $\textit{weight}(S) := \sum_{x\in S} \textit{weight}(x)$ and $\textit{profit}(S) := \sum_{x\in S} \textit{profit}(x)$. The goal is now to select a subset of the items whose profit is maximized, under the condition that the total weight of the selected items is at most W. From now on, we will assume that $\textit{weight}(x_i) \leqslant W$ for all i. (Items with $\textit{weight}(x_i) > W$ can of course simply be ignored.)

The case of integer profits. We will first develop an algorithm for the case where all the profits are integers. Let $P := \textit{profit}(X)$, that is, P is the total profit of all items. The running time of our algorithm will depend on n and P. Since P can be arbitrarily large, the running time of our algorithm will not necessarily be polynomial in n. In the next section we will then show how to obtain an FPTAS for Knapsack that uses this algorithm as a subroutine.

Our algorithm for the case where all profits are integers is a dynamic-programming algorithm. For $1 \leqslant i \leqslant n$ and $0 \leqslant p \leqslant P$, define

$$A[i,p] = \min\{\textit{weight}(S) : S \subset \{x_1,\dots,x_i\} \text{ and } \textit{profit}(S) = p\}.$$

In other words, $A[i,p]$ denotes the minimum possible weight of any subset S of the first i items such that $\textit{profit}(S)$ is exactly p. When there is no subset $S \subset \{x_1,\dots,x_i\}$ of profit exactly p then we define $A[i,p] = \infty$. Note that Knapsack asks for a subset of weight at most W with the maximum profit. This maximum profit is given by $\textsc{opt} := \max\{p : 0 \leqslant p \leqslant P \text{ and } A[n,p] \leqslant W\}$. This means that if we can compute all values $A[i,p]$ then we can compute $\textsc{opt}$. From the table A we can then also compute a subset S such that $\textit{profit}(S) = \textsc{opt}$—see below for details. As is usual in dynamic programming, the values $A[i,p]$ are computed bottom-up by filling in a table. It will be convenient to extend the definition of $A[i,p]$ to include the case $i = 0$, as follows: $A[0,0] = 0$ and $A[0,p] = \infty$ for $p > 0$. Now we can give a recursive formula for $A[i,p]$.

Lemma 9.1

$$A[i,p] = \begin{cases} 0 & \textit{if } p = 0 \\ \infty & \textit{if } i = 0 \textit{ and } p > 0 \\ A[i-1,p] & \textit{if } i > 0 \textit{ and } 0 < p < \textit{profit}(x_i) \\ \min(A[i-1,p], A[i-1,p-\textit{profit}(x_i)] + \textit{weight}(x_i)) & \textit{if } i > 0 \textit{ and } p \geqslant \textit{profit}(x_i) \end{cases}$$

Proof. The first two cases are simply by definition. Now consider third and fourth case. Obviously the minimum weight of any subset of $\{x_1,\dots,x_i\}$ of total profit p is given by one of the following two possibilities:

- the minimum weight of any subset $S \subset \{x_1,\dots,x_i\}$ with profit p and $x_i \in S$, or
- the minimum weight of any subset $S \subset \{x_1,\dots,x_i\}$ with profit p and $x_i \notin S$.

In the former case, $\textit{weight}(S)$ is equal to $\textit{weight}(x_i)$ plus the minimum weight of any subset $S \subset \{x_1,\dots,x_{i-1}\}$ with profit $p - \textit{profit}(x_i)$, which is given by $A[i-1, p-\textit{profit}(x_i)]$. (This is also correct when $A[i-1, p-\textit{profit}(x_i)] = \infty$. In that case there is no subset $S \subset \{x_1,\dots,x_{i-1}\}$ of profit $p - \textit{profit}(x_i)$, so there is no subset $S \subset \{x_1,\dots,x_i\}$ of profit p that includes x_i.) In the latter case, $\textit{weight}(S)$ is equal to the minimum weight of any subset $S \subset \{x_1,\dots,x_{i-1}\}$ with profit p, which is $A[i-1,p]$. (Again, this is also correct when $A[i-1,p] = \infty$.) When $p < \textit{profit}(x_i)$ the former possibility does not apply, which proves the lemma for the third case. Otherwise we have to take the best of the two possibilities, proving fourth case. □

Based on this lemma, we can immediately give a dynamic-programming algorithm.

Algorithm *IntegerWeightKnapsack*(X, W)

```
1.  Let $A[0..n, 0..P]$ be an array, where $P = \sum_{i=1}^{n} \mathit{profit}(x_i)$.
2.      $A[0,0] \leftarrow 0$
3.      for $p \leftarrow 1$ to $P$
4.          do $A[0,p] \leftarrow \infty$
5.      for $i \leftarrow 1$ to $n$
6.          do for $p \leftarrow 1$ to $P$
7.                  do if $\mathit{profit}(x_i) \leqslant p$
8.                      then $A[i,p] \leftarrow \min(A[i-1,p], \mathit{weight}(x_i)+A[i-1, p-\mathit{profit}(x_i)])$
9.                      else $A[i,p] \leftarrow A[i-1,p]$
10.     $\text{OPT} \leftarrow \max\{p : 0 \leqslant p \leqslant P \text{ and } A[n,p] \leqslant W\}$
11.     Using the table $A$, find a subset $S \subset X$ of profit OPT and total weight at most $W$.
12.     return $S$
```

Finding an optimal subset S in line 11 of the algorithm can be done by "walking back" in the table A, as is standard in dynamic-programming algorithms—see also the chapter on dynamic programming from [CLRS]. For completeness, we describe a subroutine *ReportSolution* that finds an optimal subset.

Algorithm *ReportSolution*(X, A, OPT)

```
1.  $p \leftarrow \text{OPT}$; $S \leftarrow \emptyset$
2.      for $i \leftarrow n$ downto 1
3.          do if $\mathit{profit}(x_i) \leqslant p$
4.              then if $\mathit{weight}(x_i) + A[i-1, p-\mathit{profit}(x_i)] < A[i-1,p]$
5.                  then $S \leftarrow S \cup \{x_i\}$; $p \leftarrow p - \mathit{profit}(x_i)$
6.      return $S$
```

It is easy to see that *IntegerWeightKnapsack*, including the subroutine *ReportSolution*, runs in $O(nP)$ time. We get the following theorem.

Theorem 9.2 *Suppose all profits in a* Knapsack *instance are integers. Then the problem can be solved in $O(nP)$ time, where $P := \mathit{profit}(X)$ is the total profit of all items.*

An FPTAS for Knapsack. How can we use the result above to obtain an FPTAS for the general case, where the profits can be arbitrarily large and need not even be integers? For this we would need to scale the profits down so that they are not too large, and then round them so that they are integral. This scaling and rounding will introduce some "error" in the computations—that is, since we will not work with the exact profits anymore, we may erroneously believe that a certain subset is better than another subset. The goal is to do the scaling and rounding in such a way that this error is small so that the result will be close to optimal. To obtain a $(1-\varepsilon)$-approximation, we want the error to be at most $\varepsilon \cdot \text{OPT}$. This leads to the following idea.

Suppose we "round" every *profit*(x_i) to the next larger multiple of $(\varepsilon/n) \cdot \text{OPT}$. Since there are no more than n items in any subset S, such a rounding cannot incur an error of more than $\varepsilon \cdot \text{OPT}$ in the profit of S. The nice thing is that after this rounding the whole problem can be scaled, because every profit is now a multiple of $(\varepsilon/n) \cdot \text{OPT}$. This suggests to replace each *profit*(x_i) by p_i, where p_i is the smallest integer such that $\mathit{profit}(x_i) \leqslant p_i \cdot ((\varepsilon/n) \cdot \text{OPT})$.

The value p_i satisfying this condition is given by

$$p_i := \lceil \frac{\mathit{profit}(x_i)}{(\varepsilon/n)\cdot \textsc{opt}} \rceil. \tag{9.1}$$

Okay, we have replaced the profits $\mathit{profit}(x_i)$ by integer profits p_i. How large can the integers p_i be? Let j be such that x_j is an item of maximum profit, that is, j is such that $\mathit{profit}(x_i) \leqslant \mathit{profit}(x_j)$ for all $1 \leqslant i \leqslant n$. Then we also have $p_i \leqslant p_j$ for all i. Obviously, $\textsc{opt} \geqslant \mathit{profit}(p_j)$. Hence,

$$p_j \leqslant \lceil \frac{\mathit{profit}(x_j)}{(\varepsilon/n)\cdot \mathit{profit}(x_j)} \rceil = \lceil n/\varepsilon \rceil.$$

It seems we are in business: we have transformed the problem to a problem where all the profits are integers in a polynomial range, namely $1..\lceil n/\varepsilon\rceil$, in such a way that the error introduced by the transformation is not too large. There is one problem, however: we do not know $\textsc{opt}$, so we cannot round the profits using (9.1). Therefore, instead of using $\textsc{opt}$ we use the lower bound $\textsc{lb} := \max_i \mathit{profit}(x_i)$, and instead of using the profits p_i we use $\mathit{profit}^*(x_i)$ which is defined as

$$\mathit{profit}^*(x_i) := \lceil \frac{\mathit{profit}(x_i)}{(\varepsilon/n)\cdot \textsc{lb}} \rceil. \tag{9.2}$$

Note that the profits are still integers in the range $1..\lceil n/\varepsilon\rceil$. Our FPTAS now look as follows.

Algorithm *Knapsack-FPTAS*(X, W, ε)
```
1.  $\textsc{lb} \leftarrow \max_{1 \leqslant i \leqslant n} \mathit{profit}(x_i)$.
2.      For all $1 \leqslant i \leqslant n$, let $\mathit{profit}^*(x_i) \leftarrow \lceil \frac{\mathit{profit}(x_i)}{(\varepsilon/n)\cdot\textsc{lb}} \rceil$.
3.      Compute a subset $S^* \subset X$ of maximum profit and total weight at most $W$ using
        algorithm IntegerWeightKnapsack, using the new profits $\mathit{profit}^*(x_i)$ instead of
        $\mathit{profit}(x_i)$.
4.      return $S^*$
```

We conclude with the following theorem.

Theorem 9.3 *Knapsack-FPTAS computes in $O(n^3/\varepsilon)$ time a subset $S^* \subset X$ of weight at most W whose profit is at least $(1-\varepsilon)\cdot \textsc{opt}$, where $\textsc{opt}$ is the maximum profit of any subset of weight at most W.*

Proof. To prove the running time, we observe that $\mathit{profit}^*(x_i) \leqslant \lceil n/\varepsilon \rceil$ for all $1 \leqslant i \leqslant n$. Hence, the total profit $\mathit{profit}^*(X)$ is at most $n \cdot \lceil n/\varepsilon\rceil$, so by Theorem 9.2 the algorithm runs in $O(n^3/\varepsilon)$ time.

To prove the approximation ratio, let S_{opt} denote an optimal subset, that is, a subset of weight at most W such that $\mathit{profit}(S_{\text{opt}}) = \textsc{opt}$. Let S^* denote the subset returned by the algorithm. Since we did not change the weights of the items, the subset S^* has weight at most W. It remains to show that $\mathit{profit}(S^*) \geqslant (1-\varepsilon)\cdot \textsc{opt}$.

Because S^* is optimal for the new profits, we have $\mathit{profit}^*(S^*) \geqslant \mathit{profit}^*(S_{\text{opt}})$. Moreover

$$\frac{\mathit{profit}(x_i)}{(\varepsilon/n)\cdot \textsc{lb}} \leqslant \mathit{profit}^*(x_i) \leqslant \frac{\mathit{profit}(x_i)}{(\varepsilon/n)\cdot \textsc{lb}} + 1.$$

Hence, we have

$$\begin{array}{rcl}
\mathit{profit}(S^*) & = & \sum_{x_i \in S^*} \mathit{profit}(x_i) \\
& \geqslant & \sum_{x_i \in S^*} (\varepsilon/n) \cdot \textsc{lb} \cdot (\mathit{profit}^*(x_i) - 1) \\
& \geqslant & (\varepsilon/n) \cdot \textsc{lb} \cdot \sum_{x_i \in S^*} \mathit{profit}^*(x_i) - |S^*| \cdot (\varepsilon/n) \cdot \textsc{lb} \\
& \geqslant & (\varepsilon/n) \cdot \textsc{lb} \cdot \sum_{x_i \in S_{\text{opt}}} \mathit{profit}^*(x_i) - (|S^*|/n) \cdot \varepsilon \cdot \textsc{lb} \\
& \geqslant & \sum_{x_i \in S_{\text{opt}}} \mathit{profit}(x_i) - \varepsilon \cdot \textsc{lb} \\
& \geqslant & \textsc{opt} - \varepsilon \cdot \textsc{lb} \\
& \geqslant & \textsc{opt} - \varepsilon \cdot \textsc{opt}
\end{array}$$

It follows that $\mathit{profit}(S^*) \geqslant (1 - \varepsilon) \cdot \textsc{opt}$, as claimed.

□